Scalable Algorithms in Nonparametric Computational Statistics

Johannes Graner
Supervisor: Raazesh Sainudiin

September 12, 2022

Abstract

Three problems in distributed computing and non-parametric computational statistics are explored to develop scalable distributed algorithms. The problems are rigorous global optimization, trend detection in time series, and non-parametric density estimation with tree-based histograms. Scalable algorithms are implemented for each problem using the fault-tolerant distributed framework of Apache Spark.

Contents

0.1 Introduction

With the advent of powerful computers, many previously infeasible approaches to statistics are now commonplace. Methods such as bootstrapping, Monte Carlo methods, and kernel density estimation are very computationally intensive but have led to great advances in human knowledge. The rapid advances in computer technology have allowed these techniques to become more powerful each time better computer hardware is developed. However, there are limits on how powerful a single computer can be, and many methods in computational statistics are limited by the performance of that single computer.

One approach to circumventing this problem is to adapt the statistical methods to efficiently use more than a single machine in the computations. This is called distributed computing and introduces several challenges, but can, if implemented correctly and efficiently, easily be scaled to use an arbitrary number of comparatively weak computers with a total computing power much greater than any single machine. Several computers working on the same problem in a distributed fashion is called a *cluster* and each computer is referred to as a *worker*.

The main challenge in distributed computing is that any one worker can only access its own memory, meaning that the state of the entire computation is usually not accessible to a worker. For example, in density estimation each worker only has access to a small part of the data set, and the algorithm must be adjusted accordingly. The plan for this thesis is to explore and implement distributed (also called *scalable*) algorithms for three different problems.

The first problem is global optimization, i.e. finding global extrema for a real-valued function. This is an important problem in many fields of mathematics, including e.g. maximum likelihood estimation in statistics. The problem is approached using rigorous computation and interval arithmetic.

The second problem is finding summary statistics of time series. It can be very difficult to intuitively find and understand features of long-running time series, so summary statistics are used to extract such features. One such feature is the trend (long- or short-term) of the time series, which this thesis focuses on.

The third and final problem considered in the thesis is non-parametric density estimation. When analyzing a data set, estimating the underlying distribution of the data can be very useful, but often no assumptions can reasonably be made on what this distribution might be. Histograms are a well-known class of non-parametric density estimators, and this thesis explores a scalable tree-based histogram estimator for arbitrary densities.

Chapter 1

Scalable Global Optimization

1.1 Rigorous Computation

In many scientific domains, measurements are known not only to be inexact, it is also known how inexact they are. Such measurement errors accumulate when many measurements are used together in a computation. In order to know the error in the final result, a systematic way of accumulating the errors is needed.

In fact, all floating point operations done on a computer have inherent errors, usually called round-off error.

1.1.1 Floating Point Systems

Real numbers generally have infinite expansions regardless of base (binary, decimal, etc.). Since computers do not have infinite memory, most real numbers cannot be perfectly represented in a computer, instead having an approximate, finite representation in a *floating point system*.

A number in a floating point system is in the form

$$x = \pm m \cdot b^e = \pm 0.m_1 m_2 \dots m_l \cdot b^e \tag{1.1}$$

where the base b is an integer $b \geq 2$, m is the mantissa of length l, consisting of integers $1 \leq m_1 \leq b-1$ and $0 \leq m_k \leq b-1$ for $k = 2, \dots, l$, and e is an integer such that $e_{min} \leq e \leq e_{max}$. A floating point system is defined as $R(b, l, e_{min}, e_{max})$ and this forms a screen for which numbers are representable in the system.

To represent a real number x in a floating point system R, the number is rounded to a number that is representable in the system. The rounding can be upwards (to the smallest number in R that is larger than x), downwards (to the largest number in R that is smaller than x), or to the nearest number in R. In this way, R forms a screen for the real numbers.

The result of any operation (e.g. multiplication) must also be rounded to be representable in the floating point system. If a large number of such operations are carried out, significant round-off error can accumulate and yield a final result that is unusable [12].

For a more in-depth description of floating point systems and their arithmetic, see e.g. [3].

1.1.2 Interval Arithmetic

Both measurement errors and round-off errors can be carried forward properly through computations using interval arithmetic.

An interval $\boldsymbol{x} = [\underline{x}, \overline{x}] \subset \mathbb{R}$ can be used to encapsulate the error by using the lower and upper bounds for the number or measurement as the lower and upper bound of the interval, respectively. The set of all real intervals is $\mathbb{IR} = \{[\underline{x}, \overline{x}] : \underline{x} \leq \overline{x},\ \underline{x}, \overline{x} \in \mathbb{R}\}$. The diameter of $\boldsymbol{x}$ is $d(\boldsymbol{x}) = \overline{x} - \underline{x}$ and the midpoint is $\operatorname{mid}(\boldsymbol{x}) = \frac{\underline{x}+\overline{x}}{2}$. The largest absolute value of $\boldsymbol{x}$ is $|\boldsymbol{x}| = \max\{|x| : x \in \boldsymbol{x}\} = \max\{|\underline{x}|, |\overline{x}|\}$ and the smallest absolute value is $\langle \boldsymbol{x} \rangle = \min\{|x| \in \boldsymbol{x}\} = \begin{cases} 0, & 0 \in \boldsymbol{x} \\ \min\{|\underline{x}|, |\overline{x}|\}, & 0 \notin \boldsymbol{x} \end{cases}$. The absolute value of $\boldsymbol{x}$ is $|\boldsymbol{x}|_{[]} = [\langle \boldsymbol{x} \rangle, |\boldsymbol{x}|]$. A useful measure is the relative diameter of an interval, denoted $d_{rel}(\boldsymbol{x})$. If $0 \notin \boldsymbol{x}$, then $d_{rel}(\boldsymbol{x}) = \frac{d(\boldsymbol{x})}{\langle \boldsymbol{x} \rangle}$, otherwise $d_{rel}(\boldsymbol{x}) = d(\boldsymbol{x})$. An interval with $\underline{x} = \overline{x}$ is called *thin*. An interval that is not thin, i.e. $\underline{x} < \overline{x}$, is called *thick*.

Intervals are not restricted to the real line, in particular they can be extended to interval vectors in $\mathbb{IR}^d$ by letting each component be an interval. The diameter, midpoint, etc. are then vectors in $\mathbb{R}^d$ and the maximal diameter of $\boldsymbol{x} \in \mathbb{IR}^d$ is defined by $d_\infty(\boldsymbol{x}) = \max_i d(\boldsymbol{x}_i)$. The maximal relative diameter is similarly defined as $d_{rel,\infty}(\boldsymbol{x}) = \max_i d_{rel}(\boldsymbol{x}_i)$.

A metric g can be defined on $\mathbb{IR}$ as $g(\boldsymbol{x}, \boldsymbol{y}) = max\{|\underline{x} - \underline{y}|, |\overline{x} - \overline{y}|\}$, yielding that $\mathbb{IR}$ is a complete metric space under g. The sequence of intervals $\{\boldsymbol{x}^{(i)}\}$ converging to $\boldsymbol{x}$ is equivalent to $\underline{x}^{(i)} \to \underline{x}$ and $\overline{x}^{(i)} \to \overline{x}$. This can also be extended to $\mathbb{IR}^d$ and hence it is reasonable to consider continuous and differentiable functions $f : \mathbb{IR}^d \to \mathbb{IR}^k$ with continuity and differentiability defined as usual with the metric g.

With $\mathbb{IR}$ as a metric space, an arithmetic is defined by the operations $\{+, -, \cdot, /\}$. Let $\circ \in \{+, -, \cdot, /\}$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR}$. Assume $0 \notin \boldsymbol{y}$ when $\circ = /$. Then, $\boldsymbol{x} \circ \boldsymbol{y}$ is defined by $\boldsymbol{x} \circ \boldsymbol{y} = \{x \circ y : x \in \boldsymbol{x}, y \in \boldsymbol{y}\}$. Using this definition of the arithmetic operations makes them *inclusion isotonic*; if $\boldsymbol{v} \subseteq \boldsymbol{x}$ and $\boldsymbol{w} \subseteq \boldsymbol{y}$, then $\boldsymbol{v} \circ \boldsymbol{w} \subseteq \boldsymbol{x} \circ \boldsymbol{y}$. This is due to $\boldsymbol{x}$ and $\boldsymbol{y}$ being simply connected compact intervals, and $\circ$ a continous operation. With these properties, $\boldsymbol{x} \circ \boldsymbol{y}$ contains the minimum, maximum and all values in between of $x \circ y$ where $x \in \boldsymbol{x}, y \in \boldsymbol{y}$, i.e. $\boldsymbol{x} \circ \boldsymbol{y} = [\min\{x \circ y\}, \max\{x \circ y\}]$.

The binary operations of $+$ and $\cdot$ have identity elements, the thin intervals $[0, 0]$ and $[1, 1]$, respectively. However, neither addition nor subtraction have inverses except when $\boldsymbol{x}$ is thin, due to $[0, 0] \subsetneq \boldsymbol{x} - \boldsymbol{x}$ and $[1, 1] \subsetneq \boldsymbol{x}/\boldsymbol{x}$ whenever

$\underline{x} < \overline{x}$. Both $+$ and $\cdot$ are associative and commutative, but only subdistributive, i.e. $\boldsymbol{x} \cdot (\boldsymbol{y} + \boldsymbol{z}) \subseteq (\boldsymbol{x} + \boldsymbol{y}) \cdot (\boldsymbol{x} + \boldsymbol{z})$.

If $f : \mathbb{R}^d \to \mathbb{R}$ is a function, then the image of f over an interval vector $\boldsymbol{x}$ is $f(\boldsymbol{x}) = \{f(x) : x \in \boldsymbol{x}\}$. Further, if f is a combination of elementary functions (sin, cos, exp, etc.), arithmetic operations, and constants, then f is inclusion isotonic and can be extended to its *natural interval extension* $F : \mathbb{IR}^d \to \mathbb{IR}$ by replacing all components of f with their interval counterparts. Since F is inclusion isotonic, for each $x \in \boldsymbol{x}$, $f(x) \in F(\boldsymbol{x})$. Since this implication is only one way, in general $F(\boldsymbol{x}) \not\subseteq f(\boldsymbol{x})$, and hence $F(\boldsymbol{x})$ is generally an overestimation of $f(\boldsymbol{x})$. However, as $d_\infty(\boldsymbol{x})$ decreases to 0, the difference between $F(\boldsymbol{x})$ and $f(\boldsymbol{x})$, $g(F(\boldsymbol{x}), f(\boldsymbol{x}))$ decreases linearly to 0 as well. In other words, by partitioning $\boldsymbol{x}$ as $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$, a better approximation of $f(\boldsymbol{x})$ is achieved by $\bigcup_i F(\boldsymbol{x}^{(i)})$.

An even better approximation is available when f is differentiable everywhere in $\boldsymbol{x}$. The *centered form* of f is based on the mean value theorem.

$$\begin{aligned} f(x) =& f(c) + \nabla f(b) \cdot (x - c) \in f(c) + \nabla f(\boldsymbol{x}) \cdot (x - c) \subseteq \\ \subseteq& f(c) + \nabla F(\boldsymbol{x}) \cdot (\boldsymbol{x} - c) = F_c(\boldsymbol{x}) \end{aligned}$$

where $b, c, x \in \boldsymbol{x}$ with $c < b < x$ and real numbers are treated as thin intervals where necessary. $g(F_c(\boldsymbol{x}), f(\boldsymbol{x}))$ decreases to 0 quadratically as $d_\infty(\boldsymbol{x}) \to 0$.

Another way to improve the approximation $F(\boldsymbol{x})$ is to simplify the expression for $f(x)$ in order to minimize the number of occurances of x, and use thin intervals for constants and parameters wherever possible.

For an in-depth treatment of interval arithmetic, see e.g. [3].

1.2 Global Optimization

Global optimization refers to finding the global minimum or maximum value of a function $f : \mathbb{R}^d \to \mathbb{R}$ in a specified domain $\boldsymbol{X} \in \mathbb{R}^d$. Without loss of generality, the minimum is sought, since the maximum is found by minimizing $-f$. In order to use the advantages of interval arithmetic, some assumptions on f are necessary. The most basic assumptions are i) f has an inclusion isotonic interval extension F, ii) $\boldsymbol{X}$ is compact, and iii) f is well-defined everywhere in $\boldsymbol{X}$.

The domain $\boldsymbol{X} = \left(\boldsymbol{X}^{(1)}, \ldots, \boldsymbol{X}^{(n)}\right) = \left([\underline{X}^{(1)}, \overline{X}^{(1)}], \ldots, [\underline{X}^{(n)}, \overline{X}^{(n)}]\right)$ is then partitioned component-wise as $\boldsymbol{X}^{(j)} = \bigcup_{i=1}^{b} [x_i^{(j)}, x_{i+1}^{(j)}]$, where $x_1^{(j)} = \underline{X}^{(j)}$, $x_{b+1}^{(j)} = \overline{X}^{(j)}$, and $i < k \implies x_i^{(j)} < x_k^{(j)}$. In other words, $\boldsymbol{X}$ is divided into b thick intervals $\boldsymbol{x}_i = (\boldsymbol{x}_i^{(1)}, \ldots, \boldsymbol{x}_i^{(n)}), i = 1, \ldots, b$ that overlap only at the boundaries. The general approach is to test if a global minimum can occur in one of the intervals in the partition, and discard the interval if such is not the case. The remaining intervals in the partition are then bisected along the axis of maximum diameter and the process is repeated until all remaining intervals have (maximum) relative diameter no larger than a tolerance ϵ. Let $\boldsymbol{F}_i = F(\boldsymbol{x}_i)$. Four

such tests are used, along with a verification process, all of which are explained in detail in [3].

1.2.1 Midpoint cut-off test

This test requires only the basic assumptions on f and consists of comparing the value of f at some point to the lower bounds $\underline{F}_i$. If $x \in \boldsymbol{X} \backslash \boldsymbol{x}_i$ and $f(x) < \underline{F}_i$, then the global minimum of f cannot possibly be in $\boldsymbol{x}_i$ since there is a point $x \notin \boldsymbol{x}_i$ with $f(x) < f(x')$ for all $x' \in \boldsymbol{x}_i$.

The most promising interval to check for a test point is the interval with minimal $\underline{F}_i$, and since there is no way to know which point in $\boldsymbol{x}_i$ yields the minimal value, the midpoint is chosen for simplicity.

Hence, the midpoint cut-off test is performed as follows, given $\{\boldsymbol{x}_i\}_{i=1}^{b}$.

1. Find the index $k = \operatorname{argmin}_i \underline{F}_i$ that corresponds to the best guess of minimizing interval.
2. Find the midpoint $c = \operatorname{mid} \boldsymbol{x}_k$.
3. Discard all intervals $\boldsymbol{x}_j$ where $\underline{F}_j > \underline{F}([c, c])$.

1.2.2 Monotonicity test

The monotonicity test requires that f is continuously differentiable everywhere in $\boldsymbol{X}$, and determines if f is strictly monotone over a whole interval. A strictly monotone function over a compact interval must have its minimum value on the boundary of said interval. If $\boldsymbol{x}$ is such an interval contained in the interior of $\boldsymbol{X}$, then $\boldsymbol{x}$ can be discarded since there is an adjacent box that is guaranteed to contain a point with lower function value than any point in $\boldsymbol{x}$. If $\boldsymbol{x}$ intersects the boundary of $\boldsymbol{X}$, then $\boldsymbol{x}$ can only be discarded if this intersection does not contain $\underline{F}(\boldsymbol{x})$, which can be determined from the monotonicity.

The test is as follows, given $\boldsymbol{x}$, $\boldsymbol{X}$ and $\nabla F(\boldsymbol{x})$.

- For $i = 1, \ldots, n$:
  1. If $0 \in \nabla F(\boldsymbol{x})_i$, go to next index.
  2. Otherwise, f is monotone in at least one component over $\boldsymbol{x}$ and hence $\boldsymbol{x}$ can be removed unless $\boldsymbol{x} \cap \boldsymbol{X} \neq \emptyset$, in which case the following is checked.
     - If $\min \nabla F(\boldsymbol{x})_i > 0$ and $\underline{X}^{(i)} = \underline{x}^{(i)}$, then modify $\boldsymbol{x}$ to be the thin interval $[\underline{x}^{(i)}, \underline{x}^{(i)}]$.
     - Else, if $\max \nabla F(\boldsymbol{x})_i < 0$ and $\overline{X}^{(i)} = \overline{x}^{(i)}$, then modify $\boldsymbol{x}$ to be the thin interval $[\overline{x}^{(i)}, \overline{x}^{(i)}]$.
     - Else, remove $\boldsymbol{x}$ from the search and stop the iteration.

1.2.3 Concavity test

This test requires that f is twice continuously differentiable. If $\boldsymbol{x}$ is contained in the interior of $\boldsymbol{X}$, then f can only have its global minimum in $\boldsymbol{x}$ if f is concave over $\boldsymbol{x}$. A necessary condition for concavity is that the Hessian $\nabla^2 f$ is positive semi-definite, which in turn has the necessary condition that all diagonal elements of $\nabla^2 f$ are non-negative.

The test is as follows, given $\boldsymbol{x} \subseteq \operatorname{int}(\boldsymbol{X})$ and the diagonal elements of $\nabla^2 F(\boldsymbol{x})$.

- If, for any $i = 1, \ldots, n$, $\underline{\nabla^2 F(\boldsymbol{x})} < 0$, then remove $\boldsymbol{x}$ from the search.

1.2.4 Interval Newton test

This test requires that f is twice continuously differentiable. For $\boldsymbol{x} \subseteq \operatorname{int}(\boldsymbol{X})$, a global (or local) minimum $x \in \boldsymbol{x}$ must satisfy $\nabla f(x) = 0$. This requirement can be exploited by performing one step of the Newton Gauss-Seidel method for finding zeros.

The interval extension of the method is given by computing $\boldsymbol{A} = \boldsymbol{R} \cdot \nabla^2 F(\boldsymbol{x})$ and $\boldsymbol{b} = \boldsymbol{R} \cdot \nabla f(\operatorname{mid}(\boldsymbol{x}))$, where $\boldsymbol{R} \approx (\operatorname{mid}(\nabla^2 F(\boldsymbol{x})))^{-1}$. The system $\boldsymbol{A}(\boldsymbol{x} - \operatorname{mid}(\boldsymbol{x})) = \boldsymbol{b}$ is then solved by Gauss-Seidel iteration. Let $c = \operatorname{mid}(\boldsymbol{x})$ and $N'_{GS}(\boldsymbol{x})$ be as follows.

$$
\begin{aligned}
\boldsymbol{y} &= \boldsymbol{x} \\
\boldsymbol{y}_i &= \left(c_i - \boldsymbol{A}_{ii}^{-1} \left(\boldsymbol{b}_i + \sum_{j=1, j \neq i}^{n} \boldsymbol{A}_{ij} (\boldsymbol{y}_j - c_j) \right) \right) \cap \boldsymbol{y}_i, \; i = 1, \ldots, n \\
N'_{GS}(\boldsymbol{x}) &= \boldsymbol{y}
\end{aligned}
$$

The test is then as follows, given $\boldsymbol{x} \subseteq \operatorname{int}(\boldsymbol{X})$ and $N'_{GS}(\boldsymbol{x})$.

- If $N'_{GS}(\boldsymbol{x})$ is empty, then $0 \notin \nabla F(\boldsymbol{x})$ and $\boldsymbol{x}$ is discarded from the search.
- If $N'_{GS}(\boldsymbol{x}) \subseteq \operatorname{int}(\boldsymbol{x})$, then $\boldsymbol{x}$ is replaced by the contracted interval $N'_{GS}(\boldsymbol{x}) \cap \boldsymbol{x}$.
- If $0 \in \boldsymbol{A}_{ii}$, and $\boldsymbol{x}_i$ is split into a union of two non-empty intervals $\boldsymbol{x}_i^1$ and $\boldsymbol{x}_i^2$, then the Gauss-Seidel iteration continues on $\boldsymbol{x}_i^1$, and $\boldsymbol{x}_i^2$ is added to the list of intervals to be searched.

In total, the interval extended Newton Gauss-Seidel method can yield up to $n + 1$ new intervals to search.

1.2.5 Verification

When there are no remaining interval with maximal relative diameter greater than ϵ, the candidate intervals are checked to see if a minimum can be verified in

them. Note that this step is only possible when f is twice continuously differentiable, and is done by checking two conditions. First, if $N'_{GS}(\boldsymbol{x}) \subseteq \text{int}\,(\boldsymbol{x})$, then both existence and uniqueness of a stationary point in $\boldsymbol{x}$ is verified. Second, the positive definiteness of $\nabla^2 F(\boldsymbol{x})$ is checked to verify that the stationary point is a minimum. This is done by the following theorem, Theorem 14.1 in [3].

Theorem 1.1 *Let $\boldsymbol{H} \in \mathbb{IR}^{n\times n}$ and S be defined by $\boldsymbol{S} = I - \frac{1}{\kappa}\boldsymbol{H}$, with $d_{rel,\infty}\boldsymbol{H} \leq \kappa \in \mathbb{R}$. If $\boldsymbol{S}$ satisfies*

$$\boldsymbol{S} \cdot \boldsymbol{z} \subseteq \text{int}\,(\boldsymbol{z})$$

for an interval vector $\boldsymbol{z} \in \mathbb{IR}^n$, then $\rho(B) < 1$ for all $B \in \boldsymbol{S}$, and all symmetric matrices $A \in \boldsymbol{H}$ are positive definite.

where $\boldsymbol{H} = \nabla^2 F(\boldsymbol{x})$.

Note that this can only verify that a unique minimizer is found in the interval, there are no guarantees that a given candidate interval contains a global minimizer except in the case where there is only one candidate interval. If an interval cannot be verified to contain a minimizer, that interval should still be kept in the list of candidates since, for example, the global minimum could be a continuum of points in that interval.

1.3 Scalable Global Optimization

Interval arithmetic and the process in section 1.3 are implemented in the C++ library MRS 2.0 [15]. Although the implementation is very performant on a single machine, it is not scalable to a distributed setting with a cluster of workers. This becomes a problem when the function to be minimized is very high-dimensional, due to the vastly increased memory capacity and computational effort needed for the algorithms.

A solution to this is to use the popular distributed framework Apache Spark for handling communication and distribution of tasks in a cluster, where each worker uses MRS 2.0 for the actual work. This is accomplished by Algorithm 1.

This is less efficient than the single machine implementation for several reasons. One source of inefficiency is the communication overhead, especially since the optimization algorithm is written in C++, while Apache Spark uses Scala, meaning that data structures have to be converted between the languages between each batch. Furthermore, the nodes do not share information about the current best estimate for the minimum, meaning that a node might work on intervals that another node already knows to not contain a global minimum.

1.4 Results

Running Algorithm 1 on clusters, using worker with 2 cores and 8 GB memory each, the timings in Table 1.1 were recorded. The objective function used is the

Algorithm 1: Scalable Global Optimization

Input: $f : \mathbb{R}^d \to \mathbb{R}$, the function to be minimized.
$\boldsymbol{X}$, the domain to search.
ϵ_0, the sought tolerance level.
r, the factor to decrease ϵ by.
N, the number of nodes in the cluster.
Output: L, a list of the candidate intervals.

$\epsilon \leftarrow 1$.
Partition $\boldsymbol{X}$ into $(\boldsymbol{X}^{(1)}, \dots, \boldsymbol{X}^{(N)})$, either by using domain knowledge or, if none is available, by uniformly partitioning the first axis.
Set $L \leftarrow (\boldsymbol{X}^{(1)}, \dots, \boldsymbol{X}^{(N)})$, where L is a distributed list so that worker i has $\boldsymbol{X}^{(i)}$ in local memory.
while $\exists \boldsymbol{x} \in L : d_{rel,\infty}(\boldsymbol{x}) > \epsilon_0$ **do**
    $L_{new} \leftarrow$, distributed list.
    do in parallel
        For each $\boldsymbol{x} \in L$, compute the list of candidate intervals with tolerance ϵ and append to L_{new}.
    do in parallel
        Find $l^* = \min_{\boldsymbol{x} \in L_{new}} \overline{F(\boldsymbol{x})}$, i.e. the interval with smallest upper bound for the function values.
    do in parallel
        $L_{new} \leftarrow \{\boldsymbol{x} \in L_{new} : \underline{F(\boldsymbol{x})} \leq l^*\}$, i.e. discard intervals where the lower bound for the function is greater than the lowest upper bound.
    $L \leftarrow L_{new}$
    Re-distribute L so that the number of intervals in each node is roughly equal.
    $\epsilon \leftarrow \frac{\epsilon}{r}$
return L

Rosenbrock function given in Equation 1.2. Clearly, the speedup is negligible. A reason for this could be that the Rosenbrock function has a single global minimum at $(1, \dots, 1)$ as well as a local minimum at $(-1, 1, \dots)$ for $4 \leq d \leq 7$, where d is the domain dimension. The recorded timings use $N = 5$ and four partitions of the interval $[-10, 10]^5$.

$$f(x) = \sum_{i=1}^{d-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2, \; x \in \mathbb{R}^d \tag{1.2}$$

Table 1.2 contains the results when running the algorithm for the function in Equation 1.3, which has many sharp minima close to the global minimum $f(0) = -1$. Parameters used were $d = 3$ dimensions and $N = 4$ partitions. The initial search box was $[-0.5543245, 0.589764]^d$, chosen arbitrarily but not

No. Workers	time (s)
2	22.8
4	22.0

Table 1.1: Timings for the Rosenbrock function in 5 dimensions, using 4 partitions.

symmetric about the origin. No speedup is seen in this case either.

$$f(x) = -\prod_{i=1}^{d} \left(\cos(100\pi x_i) \exp(-\frac{x_i^2}{200}) \right)^2 \tag{1.3}$$

Table 1.2: Timings for function 1.3 in 3 dimensions, using 4 partitions.

No. Workers	time (s)
2	10.98
4	13.68

1.5 Discussion

In both cases, running the original global optimization algorithm (i.e. without distributed computing) proved to be both faster and able to handle higher dimensions than the scalable extension.

The issue is that the global optimization algorithm is extremely slow on certain intervals, such as those containing the origin, but not the global minimum, in case of the Rosenbrock function. This creates a situation where the worker assigned to the problematic interval takes much longer to finish than any of the other workers, leading to all but one worker being idle for the majority of each iteration. Addressing this issue would require either improving the original algorithm or a way to interrupt computations on other workers once a better candidate interval is found.

Unfortunately, improving the original algorithm is outside the scope of this thesis, and mid-computation communication between workers is not possible in native Apache Spark, hence another framework (e.g. MPI) would be required. There are efforts to do mid-computation communication in Apache Spark, for example in the framework Maggy [1]. However, Apache Spark was chosen due to the ease of setting up the cluster and distributing the workload. Using a more complex framework is also outside the scope of the thesis, leading to the conclusion that while it should be possible to implement a scalable global optimization routine as Algorithm 1, an efficient implementation is not presented here.

Chapter 2

Sketching with Trends

Time series often feature regions where the tendency to increase or decrease is somewhat constant. These regions can vary dramatically in scale while behaving quite similarly. Such behaviors are called *trends* and are of much interest in many disciplines, among them analysis of financial time series. The time series tends to increase when the trend is *up*, and decrease when the trend is *down*.

While such trends can be easy to spot for a human, it is not trivial how a trend should be defined precisely for use in automated systems, which are necessary when the time series is so long that a human cannot find all short-term trends in a reasonable amount of time. One possible definition of a trend which lends itself nicely to fast computation is the one used in Trend Calculus [9, 10].

2.1 Trend Calculus

Time series can fluctuate rapidly, and need to be smoothed before determining trends. The smoothing is accomplished by only considering the highest and lowest point within a window of specified size. By increasing the window size, smaller fluctuations inside the window are disregarded as noise, while the highest and lowest points are taken to be indicative of the current trend.

In the Trend Calculus algorithm, an up trend is defined as a period with "higher highs and higher lows", while a down trend has "lower highs and lower lows".

A complete description of the Trend Calculus algorithm is as follows.

2.1.1 Algorithm

The input to the algorithm is a window size $s \geq 2$ and a time series T with $N \geq 2s$ observations.

$$T = \{p_i = (t_i, x_i) \in \mathbb{R}^2 : t_i < t_j, 1 \leq i < j \leq N\}$$

To create the windows, define the function $f_s : T^s \to T^2$ as

$$f_s(p_1, \ldots, p_s) = (\max_{t_i}(\min_{x_i}\{(t_i, x_i) : i = 1, \ldots, s\}), \\ \min_{t_i}(\max_{x_i}\{(t_i, x_i) : i = 1, \ldots, s\}))$$

which finds the latest minimum and earliest maximum among the points $p_1, \ldots, p_s$.

Assume that N is divisible by s (otherwise, only consider $N - (N \mod s)$ points of the series). Let $n = \frac{N}{s} \in \mathbb{Z}_{\geq 2}$ and

$$W = \{w_k = (l_k, h_k) = f_s(p_{1+s(k-1)}, \ldots, p_{s+s(k-1)}) : k = 1, \ldots, n\}.$$

Then W is the set of n non-overlapping windows of size s that summarize the time series T by the minimum (low) l_k and maximum (high) h_k. For each $w_k \in W$, define the trend for that window as

$$\tau_k = \begin{cases} 0, & k = 1 \\ \tau(w_{k-1}, w_k), & k > 1 \end{cases}$$

where $\tau : (\mathbb{R}^2)^2 \to \{-1, 0, 1\}$ is given by

$$\tau(w_{k-1}, w_k) = \text{sign}(\text{sign} \circ \pi_x(l_k - l_{k-1}) + \text{sign} \circ \pi_x(h_k - h_{k-1}))$$

where $\pi_x : \mathbb{R}^2 \to \mathbb{R}$ is the projection $(t, x) \mapsto x$.

Once an initial trend has been identified, each point thereafter is either part of an up or a down trend. However, there may be points after the initial period that are assigned a zero trend. The following step corrects this. Let $\tilde{n} = \min\{k : \tau_k \leq 0\}$ be the index of the first window to have a non-zero trend. Note that $\tilde{n} \geq 2$ since $\tau_1 = 0$ by definition, and let

$$w_k^* = \begin{cases} w_k, & k \leq \tilde{n} \\ w_{k-\#\{\tilde{n} < j < k : t_j = 0\}}, & \tau_k \neq 0 \text{ and } k > \tilde{n} \\ (\min_x\{a, b\}, \max_x\{a, b\}), & \tau_k = 0 \text{ and } k > \tilde{n} \end{cases}$$

where $a = \max_t\{l_{k-1}, h_{k-1}\}, b = \min_t\{l_k, h_k\}$. This creates intermediate windows at every point where the trend is 0 after the first trend has been identified, and inserts them at the correct places in the sequence (w_k^*). The length of this sequence is $n^* = n + \#\{k > \tilde{n} : \tau_k = 0\}$. This sequence is then used to define the corrected trends τ_k^*.

$$\tau_k^* = \begin{cases} 0, & k < \tilde{n} \\ \tau(w_{k-1}^*, w_k^*), & k \geq \tilde{n} \text{ and } \tau(w_{k-1}^*, w_k^*) \neq 0 \\ \tau_{k-1}^*, & \text{otherwise} \end{cases}$$

If a zero trend is found with the corrected windows as well, the last trend is carried forward. This ensures that every trend after the initial period is non-zero.

The now non-zero trends are used to define trend changes, called *reversals*, of trends as $r_1 = 0$, $r_k = \text{sign}(\tau^*_{k-1} - \tau^*_k), k > 1$. Let

$$T^*(k) = \begin{cases} \emptyset, & r_k = 0 \\ \{l_{k-1}\}, & r_k = 1 \\ \{h_{k-1}\}, & r_k = -1 \end{cases}$$

If $T^*(k) \neq \emptyset$, then $T^*(k)$ is a singleton and that element is called a *reversal point*.

This is used to define a new time series $T^{(1)} = \bigcup_{k=1}^{n^*} T^*(k) \subset T$ which contains only the points of T where a trend reversal happens. The reversal points can be used to determine the trend at every point since the reversal points are the points at which the trend changes. The trend at a point p_i is

$$\text{trend}(p_i) = \begin{cases} 0, & t_i < t_j \text{ for all } p_j \in T^{(1)} \\ \tau^*(\tilde{n})(-1)^{\#\{p_j \in T^{(1)} : t_j \leq t_i\}}, & \text{otherwise} \end{cases}$$

i.e. the trend starts at 0 and switches sign at the reversal points.

Multiple iterations

The output of the algorithm, $T^{(1)}$, is then used as input for another iteration with the same window size to get $T^{(2)} \subset T^{(1)}$. This is repeated until $\#T^{(m)} < s$, at which point $T^{(m^*)} = \emptyset \; \forall m^* > m$ and a decreasing sequence of sets $\mathcal{T}$ is given by $\mathcal{T} = (T, T^{(1)}, \ldots, T^{(m)})$, where $T \supset T^{(1)} \supset \cdots \supset T^{(m)}$. The *order* of a reversal point p is defined as $\rho(p) = \max\{k : p \in T^{(k)}\}$. A point p that is not a reversal point, i.e. $p \in T \backslash T^{(1)}$, has reversal order 0, $\rho(p) = 0$.

Since $\mathcal{T}$ is a decreasing sequence of sets, the original time series T can be augmented with information about every reversal that has occurred thus far. Each point in the augmented time series T_{tc} is $(t_i, x_i, \rho_i) \in \mathbb{R}^2 \times \{-m, \ldots, m\}$, where $\rho_i = \text{trend}(p_i) \cdot \rho(p_i)$.

If a point is a reversal of order r, it is augmented with r if it is a local minimum and $-r$ if it is a local maximum. If a point is not a reversal, it will be augmented with 0.

Since $T^{(k)} = \{p : \rho^*(p) \geq k\}$, T_{tc} contains all information necessary to recreate $\mathcal{T}$ as well as information about the type of reversal. Hence, it also contains all information necessary to determine the trend at every point of T.

2.1.2 Trend Calculus as a sketch

The Trend Calculus algorithm is easily applicable to streaming time series where each point can only be read once, and memory usage while processing is not

allowed to grow with the input size N (or at least grow very slowly compared to N). Note that almost every step in the algorithm depends only on the current and last window. The only exception is if all orders of reversals are to be computed in a single pass, in which case the last window for each reversal order must be kept in memory, as well as a buffer of reversals of lower order that will make up the next window.

A noteworthy feature of the algorithm when applied to a streaming time series is that the order of previously identified reversal points may increase over time. This is a consequence of needing two complete windows ($2s$ points) of reversal points of a given order before a higher order reversal can be identified.

A single window consists of two sets (high and low) of two floating point values (time and value), and is therefore very cheap to store. In the worst case scenario, i.e. $s = 2$ and each window contains a reversal, the number of reversal orders grows linearly with N, but in practice the number of reversal orders grow logarithmically with N [9].

Hence, $\mathcal{O}(\log N)$ need to be stored to compute all reversal orders. This grows slowly enough that it is suitable for streaming applications. As an example, a financial time series with minute-level resolution over 10 years ($N \approx 2.86 \cdot 10^6$) was shown to have 15 reversal orders [5].

2.1.3 Implementation

A commonly used framework for big data processing and distributed computing is Apache Spark. In Spark, each data point is a row consisting of fields corresponding to each dimension of the data point. As in a traditional database system, every row must follow the same schema. A time series is then a data set consisting of rows with two fields, the time and the value. In order to be able to handle several time series at once, an additional field with the label for the time series can be added. For example, in a data set with two time series, each describing the price of some commodity, the label would be the name or identifier of the commodity.

Each item in a Spark data set belongs to a partition, and all operations on the data set are done partition-wise. The partitions can reside on different workers in the cluster. In order to use the Spark framework for Trend Calculus, the algorithm must be translated into a series of operations on partitions.

Apache Spark is written in Scala and there are APIs for Scala, Java, Python, and R. The Scala API is chosen due to being particularly robust with compile-time type checking and access to all low-level functionality of Spark. Additionally, Scala supports functional programming which makes programs written in this style easy to verify.

The algorithm is implemented using `flatMapGroupsWithState`, which allows the partitions to be mapped, in order, to process some output with a low memory footprint state that is passed to the next partition. In this case, the state consists of the last window, the last trend, and any remaining data points that could not be processed due to not filling a window. In order to compute all reversal orders in a single pass, the state contains the last window and trend

for all as yet discovered reversal orders. Since the memory footprint of a window and trend are fixed, and the number of remaining points cannot exceed the window size s, the memory footprint of the state is proportional to the number of reversal orders, which is generally logarithmic in N. This is small enough that the memory footprint of the state does not become a problem, even for very large data sets.

The implementation is published under an open source licence [4].

2.1.4 Trinomial Random Walk Showcase

Define a *Trinomial Random Walk* as the discrete time stochastic process X_n given by the sequence of partial sums of independent random variables Y_i, which follow a given discrete distribution over $\{-1, 0, 1\}$, i.e. $P(Y_i = k) = p_k^{(i)},\ k \in \{-1, 0, 1\}$. Then, $X_n = \sum_{i=1}^{n} Y_i$. This results in a discrete, integer-valued time series that can, in each step, either increase by 1, decrease by 1, or stay the same.

For this showcase, Y_i are taken to be identically distributed, i.e. $p_k^{(i)} = p_k$ for all i. To further simplify the showcase, uniform probabilities are chosen, $p_{-1} = p_0 = p_1 = 1/3$.

This yields a quite simple time series where it is possible to precisely analyze the behavior of Trend Calculus. Since the higher order reversals are completely determined by the lower orders, only the first order reversals are analyzed here.

Whether a point is a trend reversal or not is determined by the window the point belongs to, the trend going into said window, as well as the previous and following windows. To keep everything simple, the minimal window size of 2 is chosen.

To characterize the reversal points, the probability of a certain sequence of reversals given a sequence of points in the time series is calculated. Since all transition probabilities are equal, all sequences of points are equally likely. The sequence length must be a multiple of the window size, here two windows (four points) are chosen as the sequence length. The number of different sequences of length n is 3^n, so for a sequence of four points, as well as the previous and following windows (eight points total), there are $3^8 = 6561$ different sequences to consider. This is sufficiently small to study analytically.

For each sequence of eight points $\{X_i\}_{i=1}^{8}$, the trend reversals are computed according to the Trend Calculus algorithm to yield $\{R_i\}_{i=1}^{8}$. Since the first and last window are not of interest, the output is $\{(X_i, R_i)\}_{i=3}^{6}$ for each of the possible sequences, where t is the trend going into the second window. The function $TC(\{X_i\}_{i=1}^{8}) = \{(X_i, R_i)\}_{i=3}^{6}$ is not injective, and since all sequences in the domain of TC are equally likely, the probability of a certain result sequence is calculated as follows.

Let $\mathcal{D} = \{\{X_i\}_{i=1}^{8} \in \mathbb{Z}^8 : X_0 = 0,\ |X_{i+1} - X_i| \leq 1\}$ be the set of all possible Trinomial Random Walks of length 8, starting at 0, and I_S be the indicator

Number of reversals	probability (%)
0	32.7
1	47.7
2	18.5
3	1.1

Table 2.1: Probabilities for different number of reversals in a trinomial random walk of length 4 with uniform transition probabilities.

function for S. Then

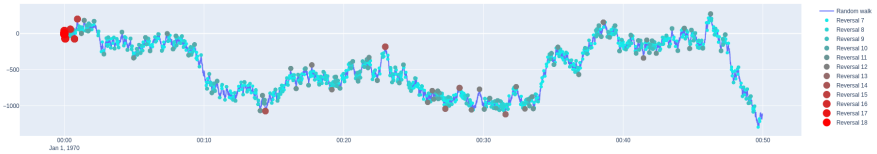
$$P(\{(X_i, R_i)\}_{i=3}^{6}) = \frac{1}{3^8} \sum_{\mathbf{X} \in \mathcal{D}} I_{\{(X_i,R_i)\}_{i=3}^{6}}(TC(\boldsymbol{X}))$$

This is used to calculate the probability for having a certain number of reversals within the inner two windows. The results of this calculation are presented in Table 2.1. The trinomial random walk is studied to contrast the trend reversals with those of a financial time series from real data. Figure 2.1 shows a few realizations of the trinomial random walk time series as well as the trends and reversal orders of oil and gold prices over the same time series length.

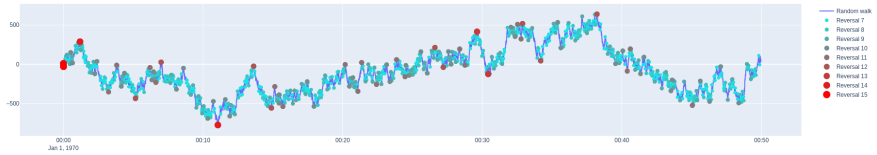
2.2 Discussion

A problem with the Trend Calculus algorithm is that it is inherently sequential since the preceeding window and trend must be known before computing the trend at a new point. Due to this, the algorithm is completely scalable in terms of memory, but not speed. Parts of the algorithm could be made scalable in computing power as well, namely the mapping of points into windows, but this is such a small part of the algorithm that the overhead for such an operation is too large in comparison to the time saved. The possibly scalable parts can without problem be parallelized locally on each worker in a shared-memory manner, but they are very difficult to do efficiently in the distributed-memory setting of the cluster. This is especially true if the sought result is the augmented time series with all original points and their reversal orders (including 0 for no reversal), since this would require a series of very expensive join operations.

One way to utilize the distributed parallelism in the cluster is to run the algorithm for several time series at the same time, using the mentioned label field to distinguish them. The `Group` part of `flatMapGroupsWithState` refers to the possibility of having several states at the same time, each corresponding to a different group, in this case different time series. This reduces the overhead compared to running the algorithm once for each time series, while also allowing more workers to do useful work at the same time. For example, the algorithm can process a selection of 10 different minute level exchange rate time series [8] ($4.5 \cdot 10^7$ data points) in 5.3 minutes, on a cluster with 4 workers, each with 16 GB memory and 4 cores.

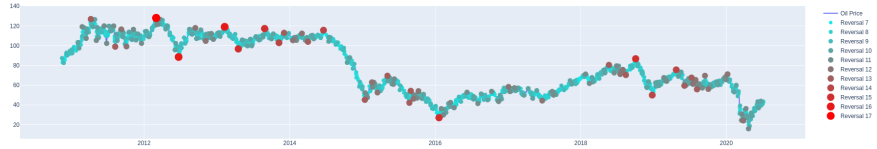


(a) Reversals in a trinomial random walk. $|T| = 3 \cdot 10^6$.

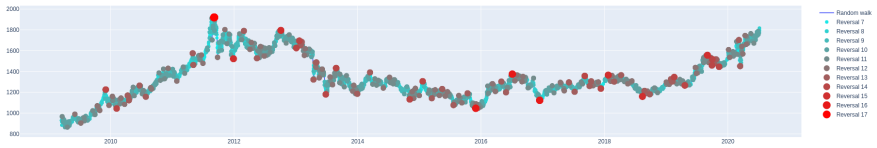


(b) Reversals in a trinomial random walk. $|T| = 3 \cdot 10^6$.

(c) Reversals in a trinomial random walk. $|T| = 3 \cdot 10^6$.



(d) Reversals in oil price (2010 - 2020). $|T| = 2.86 \cdot 10^6$.



(e) Reversals in gold price (2010 - 2020). $|T| = 3.99 \cdot 10^6$.

Figure 2.1: Trend reversals in three realizations of trinomial random walks with uniform probabilities, as well as two real financial time series. The time series lengths $|T|$ are all of the same magnitude.

The memory usage increases in proportion to the number of simultaneous computations, but since the state is small, this is not a large problem. Using the Apache Spark framework comes with the benefit that the algorithm works just as well with streaming as static data, and due to the fault-tolerance in Apache Spark, correctness is ensured even in the event of worker failures.

Chapter 3

Mapped Regular Pavings

When handling multi-dimensional data in a computer, there is a problem of how to represent the data in an efficient manner. A common family of data structures for this is trees. This family includes, among many others, binary search trees, quadtrees, and k-d trees. It is often the case that a specific type of tree is developed to suit a particular data type [7]. Mapped regular paving is a type of tree that is well suited for representing multi-dimensional data that is piece-wise constant, or can be well approximated by piece-wise constant functions. Moreover, arithmetic operation can be performed over mapped regular pavings.

A notable use-case is density estimation through histograms, where the density f for any continuous random variable can be consistently estimated given some conditions on the growth of the histogram [11], with universal performance guarantees for a given data sample of size n [2].

3.1 Regular pavings (RPs)

Consider an interval vector (or box) $\boldsymbol{x} \in \mathbb{IR}^d$. The first coordinate of maximum width is denoted ι. By splitting $\boldsymbol{x}$ perpendicularly at the mid-point along the ι-th coordinate, the child boxes $\boldsymbol{x}_L$ and $\boldsymbol{x}_R$ are given as

$$\begin{aligned}\boldsymbol{x}_L &= \boldsymbol{x}_1 \times \cdots \times [\underline{x}_\iota, \mathrm{mid}(\boldsymbol{x}_\iota)) \times \boldsymbol{x}_{\iota+1} \times \cdots \times \boldsymbol{x}_d,\\ \boldsymbol{x}_R &= \boldsymbol{x}_1 \times \cdots \times [\mathrm{mid}(\boldsymbol{x}_\iota), \overline{x}_\iota] \times \boldsymbol{x}_{\iota+1} \times \cdots \times \boldsymbol{x}_d.\end{aligned}$$

This type of bisection is called *regular*. Since the split interval in $\boldsymbol{x}_L$ is half-open, the intersection between $\boldsymbol{x}_L$ and $\boldsymbol{x}_R$ is empty. Hence, $\{\boldsymbol{x}_L, \boldsymbol{x}_R\}$ constitutes a partition of $\boldsymbol{x}$. A recursive sequence of regular bisections starting from the root box $\boldsymbol{x}_\rho$ is called a *regular paving* or n-tree [14]. A regular paving of $\boldsymbol{x}_\rho$ can be represented by a binary tree, where each node in the tree corresponds to a sub-box of $\boldsymbol{x}_\rho$. In such a tree, each node has either zero or two children. The set of leaf nodes, denoted $\mathbb{L}$, constitutes a partition of the root box $\boldsymbol{x}_\rho$ since

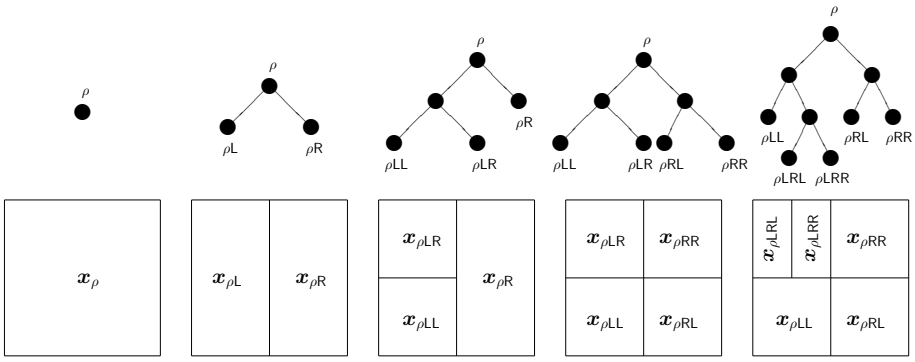


Figure 3.1: A sequence of selective bisections of boxes (nodes) along the first widest coordinate, starting from the root box (root node), produces an RP.

each box corresponding to a leaf node is an element of the partition of the box corresponding to the parent of all the leaf nodes.

Figure 3.1 gives the intuition for this, the label of each leaf node corresponding to a box in the partition. Note that the volume of a box in the partition is determined by the depth of the corresponding node in the tree. Likewise, the position of the box is determined by the position of the node compared to all other possible nodes at that depth. For example, in Figure 3.1 the node ρLL is the leftmost node at depth 2, hence the corresponding box $\boldsymbol{x}_{\rho LL}$ is in the bottom left corner since this is the box given by taking the lower interval in each split. For $x \in \boldsymbol{x}_\rho$, let $l(x) \in \mathbb{L}$ be the leaf node corresponding to the finest partitioned box that x belongs to.

Given a root box $\boldsymbol{x}_\rho$, let $\mathbb{S}_k$ be the set of all regular pavings with k splits starting from $\boldsymbol{x}_\rho$. Since $|\mathbb{L}(s)| = 1$ if $s \in \mathbb{S}_0$ and each split adds exactly one leaf, $m(s) = |\mathbb{L}(s)| = k+1$ if $s \in \mathbb{S}_k$. Let $\mathbb{S}_{i:l} = \bigcup_{k=i}^{l} \mathbb{S}_k$ be the set of regular pavings with between i and l splits, inclusive. The set of all regular pavings is denoted $\mathbb{S}_{0:\infty} = \lim_{l\to\infty} \mathbb{S}_{0:l}$.

3.1.1 Memory-efficient representation

Every node in an RP $s \in \mathbb{S}_{0:\infty}$ has zero or two children, and a node has zero children if and only if it is a leaf node. Each leaf node can also be uniquely determined by its sequence of splits from the root node, which contains – for each split – the information of whether the node belongs to the left or right child node in the split. In other words, a leaf node can be represented as a sequence of bits (0 for left, 1 for right). Such a sequence is called a *bitstring*. Using this representation, the RP is uniquely determined by the set of leaf node bitstrings. Considering that the leaves correspond to a partition of the root box, these bitstrings act as labels for the boxes in the partition.

The depth of a node l, denoted as $d(l)$, is given by the length of l's bitstring.

Denote the ancestor of l at depth d by $t(l, d)$ and note that $t(l, d)$ is given by the first d bits in l's bitstring. Hence $t(l, d)$ is the truncation of l's bitstring to its first d bits. To make $t(l, d)$ defined for all $d \in \mathbb{Z}_{>0}$, let $t(l, d) = t(l, d(l)) = l$ if $d \geq d(l)$.

This allows for efficient storage of the RP, since there is no need to keep track of the internal nodes explicitly. Representing the tree in this way differs from [7] where every node is present. Using the bitstring representation necessitates changes to the algorithms presented in [7].

Ordering

Although the memory footprint of the RP is not affected by the ordering of leaves, several operations can be made more efficient by requiring a certain order of the leaves.

A left-to-right ordering is given by $l_1 < l_2$ if and only if $t(l_1, d(l_2))$ is to the left of l_2, or $t(l_2, d(l_1))$ is to the right of l_1. In other words, to determine if $l_1 < l_2$, the two nodes are truncated to the same depth (the minimum of $d(l_1)$ and $d(l_2)$) where they can be directly compared.

For example, in Figure 3.1, $\rho LRL < \rho RL$ since $t(\rho LRL, d(\rho RL)) = = t(\rho LRL, 2) = \rho LR$ which is clearly left of ρRL.

3.1.2 Union of regular pavings

Given two RPs $s^{(1)}, s^{(2)} \in \mathbb{S}_{0:\infty}$ with the same root box $\boldsymbol{x}_\rho$, their union is given by the RP whose leaf boxes are the superimposed leaf boxes of $s^{(1)}$ and $s^{(2)}$.

This operation is given as Algorithm 1 in [7], but has to be adapted to the bitstring representation. The bitstring version of the union algorithm is given in Algorithm 2.

In the algorithm, head(s) is the first element of s, tail(s) is s with head(s) removed, $l :: s$ is the list s with element l prepended, $s_1 + s_2$ is the concatenation of lists s_1 and s_2, and $s_1 - s_2$ is s_1 with any elements from s_2 removed. Comments in all algorithms start with //.

The union operation on RPs with the same root box is associative and commutative. Additionally, if $s \in \mathbb{S}_{0:\infty}$ with root box $\boldsymbol{x}_\rho$, and $s_0 \in \mathbb{S}_0$ is the RP with root box $\boldsymbol{x}_\rho$ with no splits (the root node), then RPUnion$(s, s_0) = s =$ RPUnion(s_0, s), as s_0 is the neutral element for the union operation. Hence, $\mathbb{S}_{0:\infty}$ with a fixed root box $\boldsymbol{x}_\rho$ is a commutative monoid under the union operation.

However, if s is a sparse RP as described in Section 3.2.1, union with s_0 yields the dense representation of s. Since the dense and sparse versions of s represent the same RP in $\mathbb{S}_{0:\infty}$, this does not break the monoidal properties of $\mathbb{S}_{0:\infty}$, but it is worth considering for performance reasons due to the typically much larger memory footprint of dense RPs.

Algorithm 2: RPUnion($s^{(1)}, s^{(2)}$)

Input: $\boldsymbol{s}^{(1)}, s^{(2)} \in \mathbb{S}_{0:\infty}$ as non-empty ordered lists of leaves.
Output: s, the ordered list of leaves in the union of $s^{(1)}$ and $s^{(2)}$.

$s \leftarrow []$
$L_1 \leftarrow s^{(1)}$, $L_2 \leftarrow s^{(2)}$
while $(L_1 \neq \emptyset) \wedge (L_2 \neq \emptyset)$ **do**
    if $L_1 = \emptyset$ **then**
        $s \leftarrow s + L_2$
    else if $L_2 = \emptyset$ **then**
        $s \leftarrow s + L_1$
    else
        $l_1 \leftarrow \text{head}(L_1)$, $l_2 \leftarrow \text{head}(L_2)$
        if $l_1 = l_2$ **then**
            $s \leftarrow l_1 :: s$
            $L_1 \leftarrow \text{tail}(L_1)$, $L_2 \leftarrow \text{tail}(L_2)$
        else if l_2 is an ancestor of l_1 **then**
            $s' \leftarrow l_1 ::$ (siblings of nodes between l_1 (inclusive) and l_2 (exclusive)).
            $s' \leftarrow s'$ sorted according to left-to-right ordering.
            $s \leftarrow s + (s' - s)$
            $s \leftarrow s$ with ancestors removed
            $L_1 \leftarrow \text{tail}(L_1)$, $L_2 \leftarrow \text{tail}(L_2)$
        else if l_1 is an ancestor of l_2 **then**
            $s' \leftarrow l_2 ::$ (siblings of nodes between l_2 (inclusive) and l_1 (exclusive)).
            $s' \leftarrow s'$ sorted according to left-to-right ordering.
            $s \leftarrow s + (s' - s)$
            $s \leftarrow s$ with ancestors removed
            $L_1 \leftarrow \text{tail}(L_1)$, $L_2 \leftarrow \text{tail}(L_2)$
        else if $l_1 < l_2$ **then**
            $s \leftarrow l_1 :: s$
            $s \leftarrow s$ with ancestors removed
            $L_1 \leftarrow \text{tail}(L_1)$
        else // $l_2 < l_1$
            $s \leftarrow l_2 :: s$
            $s \leftarrow s$ with ancestors removed
            $L_2 \leftarrow \text{tail}(L_2)$
return s

3.2 Mapped regular pavings (MRPs)

Let $s \in \mathbb{S}_{0:\infty}$ be an RP with root box $\boldsymbol{x}_\rho$ and let $\mathbb{A}$ be a non-empty set. By augmenting each leaf node $l \in \mathbb{L}(s)$ with an element in $\mathbb{A}$, a piece-wise constant function $f : \mathbb{L}(s) \to \mathbb{A}$ is defined. This function can be extended to a piece-wise function $f_x : \boldsymbol{x}_\rho \to \mathbb{A}$ by first mapping $x \in \boldsymbol{x}_\rho$ to $l(x) \in \mathbb{L}$, and then applying f, i.e. $f_x = f \circ l$. This is well-defined due to s being a partition of $\boldsymbol{x}_\rho$, so x can only belong to a single leaf box. An RP s together with the set $\mathbb{A}$ and piece-wise constant function f is referred to as an $\mathbb{A}$-MRP.

Since RPs are stored as sorted lists of leaves, it is natural to store MRPs in a very similar way. Each leaf l is augmented with $f(l)$ and stored in a list $[(l_1, f(l_1)), (l_2, f(l_2)), \ldots, (l_m, f(l_m))]$. If s is an $\mathbb{A}$-MRP, let $\mathbb{L}(s)$ be the underlying RP as an ordered set of leaves. As described in [7], it is possible to define operations on MRPs with the same root box $\boldsymbol{x}_\rho$.

3.2.1 Sparse MRPs

Depending on what the MRP represents, the set $\mathbb{A}$ may contain an element that is natural to consider as a base, or neutral element. For example, let $\mathbb{A} = \{\text{True}, \text{False}\}$ and the MRP represents a discretization of some geometric object where boxes that intersect the object are given the True value and all other boxes have the False value. Then the value False could be considered as the neutral element, representing the lack of intersection with the object.

Similarly, let $\mathbb{A} = \mathbb{Z}_{\geq 0}$ and the MRP represents the number of data points that fall into each box, then the value 0 would be the base element, representing the lack of any data. If such a neutral element $e \in \mathbb{A}$ exists, the memory footprint of the MRP can be reduced significantly by only storing the leaves associated with a value other than e. An MRP in this compressed format is called a *sparse* MRP since the set of leaves stored is sparse in the the full set of leaves. An MRP that is not sparse is called *dense*.

Let $s_D = \{(l_1, a_1), \ldots, (l_m, a_m)\}$ be a dense $\mathbb{A}$-MRP with a neutral element $e \in \mathbb{A}$. The corresponding sparse $\mathbb{A}$-MRP is then $s_S = \{(l_i, a_i) \in s_D : a_i \neq e\}$. An example of a sparse MRP is given in Figure 3.2.

In order to use sparse MRPs effectively, algorithms for RP or MRP operations have to work even when all leaves are not present. Algorithms 2 and 3 are both written with sparse MRPs in mind, and work for both sparse and dense trees.

However, the result of $\text{RPUnion}(s^{(1)}, s^{(2)})$ is often much less sparse than $s^{(1)}$ or $s^{(2)}$. This is because when one RP contains a node that is an ancestor to a node in the other tree, all missing nodes in that subtree have to be included in the union. For example, if $s^{(1)} = \{\rho_L\}$, and $s^{(2)} = \{\rho_{LLL}\}$ are sparse Boolean MRPs with $e = \text{False}$, then $\text{RPUnion}(s^{(1)}, s^{(2)}) = \{\rho_{LLL}, \rho_{LLR}, \rho_{LR}\}$ in order to retain the information that both ρ_L and ρ_{LLL} were part of the original trees. This is illustrated in Figure 3.3. In this case, the union could be represented with the second operand by combining leaves associated with the same value. This is implemented as MinimiseLeaves in [7]. One could implement that algorithm for

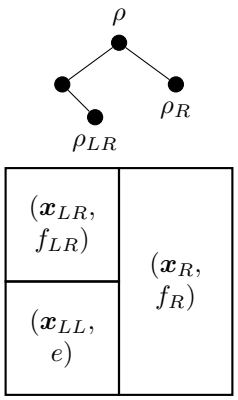


Figure 3.2: Sparse $\mathbb{A}$-MRP with neutral element $e \in \mathbb{A}$. The node ρ_{LL} is not present in the tree since $f_{LL} = e$.

this representation of MRPs, but is not done here. Figure 3.4 shows a scenario where combining leaves is not possible.

3.2.2 Operations on MRPs

The fundamental operation on $\mathbb{A}$-MRPs is described in Algorithm 3. This operation takes two $\mathbb{A}$-MRPs with the same root box and applies a function $\star : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ to each pair of overlapping leaves. MRPOperate is introduced as Algorithm 4 in [7] and is adapted for bitstring representation and sparse MRPs in Algorithm 3 below.

To accomodate sparse MRPs, the neutral element $e \in \mathbb{A}$ must also be supplied to the algorithm. If both MRPs are dense, then the neutral element argument is not used. This ensures that the same implementation works for both sparse $\mathbb{A}$-MRPs (as long as there is a neutral element in $\mathbb{A}$), and for dense $\mathbb{B}$-MRPs where $\mathbb{B}$ does not necessarily contain a neutral element. An example of MRPOperate$(s^{(1)}, s^{(2)}, +, 0)$ with $s^{(1)}, s^{(2)}$ being $\mathbb{Z}$-MRPs is shown in Figure 3.4.

3.3 MRPs as histograms

Let $x_{[n]} = (x_1, \ldots, x_n)$ be a data set with $x_i \in \mathbb{R}^d$, and let $\boldsymbol{B} \subset \mathbb{R}^d$ be an interval vector (box) such that $x_i \in \boldsymbol{B}$ for all i. Note that $\boldsymbol{x}_{[n]}$ is not necessarily a set, multiple occurrences of the same data point are allowed.

A histogram of $x_{[n]}$ is then a partition of $\boldsymbol{B}$ into $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m$ such that $\boldsymbol{b}_i \cap \boldsymbol{b}_j = \emptyset$ for all $i \neq j$ and $\bigcup_{i=1}^{m} \boldsymbol{b}_i = \boldsymbol{B}$, together with the counts $\#x_1, \ldots, \#x_m$ of the number of data points in each $\boldsymbol{b}_i$, i.e. $\#x_i = \sum_{j=1}^{n} \mathbb{1}(x_j \in \boldsymbol{b}_i)$.

Restricting the possible partitions to regular pavings of $\boldsymbol{B}$, a natural way to represent histograms is $\mathbb{Z}$-MRPs. If the dimensionality d of the data is small, a

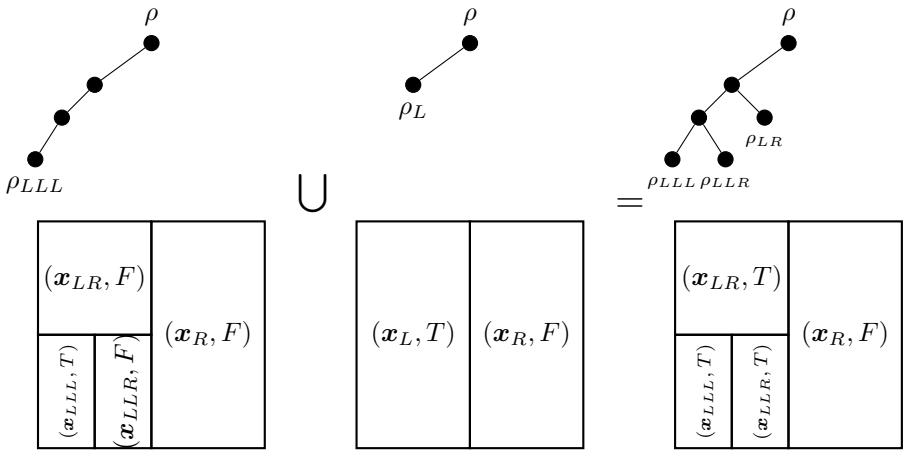


Figure 3.3: Union of the RPs belonging to two sparse Boolean MRPs with neutral element $F = \text{False}$.

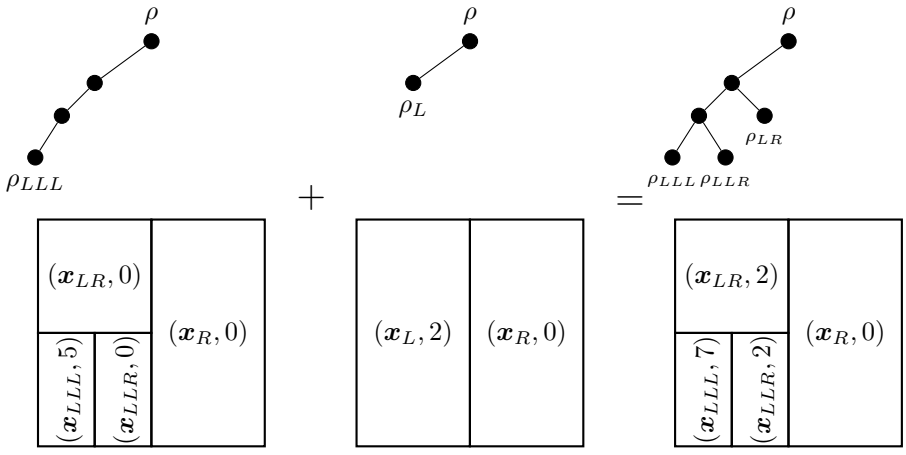


Figure 3.4: Operation on two sparse $\mathbb{Z}$-MRPs with operation $+$ and neutral element $e = 0$. Each box in the result gets augmented with the sum of the values of the boxes that overlaps with it in $s^{(1)}$ and $s^{(2)}$.

Algorithm 3: MRPOperate($s^{(1)}, s^{(2)}, \star, e$)

Input: $s^{(1)}, s^{(2)}$, $\mathbb{A}$-MRPs with the same root box,
$\star : \mathbb{A}^2 \to \mathbb{A}$,
$e \in \mathbb{A}$, the neutral element (if $s^{(1)}$ and $s^{(2)}$ are both dense, then e is not used).
Output: s, the $\mathbb{A}$-MRP with the operation applied to overlapping leaves.

$s_L \leftarrow \text{RPUnion}(\mathbb{L}(s^{(1)}), \mathbb{L}(s^{(2)}))$
$s \leftarrow []$
$L_1 \leftarrow s^{(1)}$, $L_2 \leftarrow s^{(2)}$
for $l \in s_L$ **do**
  if $\exists (l_1, v_1) \in L_1$ such that $t(l, d(l_1)) = l$ **then**
    $v_1' \leftarrow v_1$ // matching node's or ancestor's value is passed down
    if $l_1 = l$ **then**
      $i \leftarrow \text{index}((l_1, v_1), L_1)$ // index starting at 1
      $L_1 \leftarrow L_1$ with the first i elements removed
  else
    $v_1' \leftarrow e$
  if $\exists (l_2, v_2) \in L_2$ such that $t(l, d(l_2)) = l$ **then**
    $v_2' \leftarrow v_2$ // matching node's or ancestor's value is passed down
    if $l_2 = l$ **then**
      $i \leftarrow \text{index}((l_2, v_2), L_2)$ // index starting at 1
      $L_2 \leftarrow L_2$ with the first i elements removed
  else
    $v_2' \leftarrow e$
  $s \leftarrow (l, v_1' \star v_2') :: s$
return s

dense $\mathbb{Z}$-MRP can be enough. However, when d is large the histogram suffers from a curse of dimensionality since the RP requires a much larger number of splits, leading to a very large number of sub-boxes with no data ($\#x_i = 0$). A solution to this problem is to use sparse $\mathbb{Z}$-MRPs with neutral element 0.

Strategies for partitioning $\boldsymbol{B}$ into an RP based on stopping criteria to obtain minimum distance histogram esimates are presented in [13], and a scalable implementation using Apache Spark and sparse bitstring RPs is given in [14]. Here, an alternative scalable implementation is presented, one that is based on merging leaves of a sparse histogram instead of splitting leaves of a dense histogram.

Assume $x_{[n]}$ is an independent and identically distributed sample of size n from an underlying unknown density $f \in \mathcal{L}_1$, i.e. f is any density. A count based histogram of $x_{[n]}$ with $\mathbb{Z}$-MRP s can be used to estimate the density f on $\boldsymbol{B}$ by letting $f_{n,s}(x) = \frac{\#\boldsymbol{x}_{\rho\vee}}{n \operatorname{vol}(\boldsymbol{x}_{\rho\vee})}$ if $x \in \boldsymbol{x}_{\rho\vee}$ and $\boldsymbol{x}_{\rho\vee}$ is a leaf box.

The density estimate $f_{n,s}$, or f_n for notational convenience, can be seen as a sparse $\mathbb{R}$-MRP with neutral element 0. This is also a type of histogram, using frequency density of boxes instead of frequency count. When there is a need to distinguish between the two types of histograms, they are referred to as count-based or density-based, respectively.

Since a density-based histogram is a density estimate, operations that are usually performed on densities can be performed on density-based histograms as well. Two of the most important operations on densities are marginalization and finding conditional densities, as together they allow for regression based on conditional densities of the joint density estimate.

3.3.1 Marginalization

Let $f : \mathbb{R}^d \to \mathbb{R}$ be the density of a d-dimensional real-valued random variable $\boldsymbol{X} = (X_1, \ldots, X_d)$. One can marginalize $\boldsymbol{X}$ to a random variable $\boldsymbol{X}^{(m)}$ with $d' < d$ dimensions and density function $f^{(m)} : \mathbb{R}^{d'} \to \mathbb{R}$ by integrating out $x_{i_1}, \ldots, x_{i_{d-d'}}$. The marginal density $f^{(m)}$ can be computed pointwise as:

$$f^{(m)}(x) = \int \cdots \int f(x) dx_{i_1}, \ldots, dx_{i_{d-d'}}.$$

A density-based histogram can also be marginalized in a similar way, and due to the structure of RPs, marginalizing a histogram represented as an $\mathbb{R}$-MRP can be done efficiently as presented in Algorithm 4. Marginalization of MRPs is introduced in [7] and the algorithm presented here is adapted for sparse bitstring representations of MRPs.

3.3.2 Conditional density

Let $\Lambda \subset \Delta = \{1, 2, \ldots, d\}$ be a subset of coordinates of x such that $|\Lambda| = d' < d = |\Delta|$, and let $x' = (x'_j)_{j \in \Lambda}$ be a fixed subtuple of x. The non-normalized conditional density of $\boldsymbol{X}$ given x' is then $f^{|x'}\left((x_i)_{i \in \Delta \setminus \Lambda}\right) = ((x_i)_{i \in \Delta})$ where

Algorithm 4: Marginalize(s, Λ)

Input: s, an $\mathbb{R}$-MRP of a density-based histogram with root box $\boldsymbol{x}_\rho \in \mathbb{R}^d$,
Λ, the set of axes to marginalize out.
Output: s', an $\mathbb{R}$-MRP with root box $\boldsymbol{x}'_\rho \in \mathbb{R}^{d'}$, the marginalized histogram.

$s' \leftarrow s$ with all axes in Λ removed from boxes.
$s' \leftarrow s'$ grouped by box // since axes were removed, there may be duplicated boxes.
for $(x'_{\rho\vee}, L_f) \in s'$ **do** // L_f is list of densities
    oldVol $\leftarrow \mathrm{vol}\,(x_{\rho\vee})$ // Volume of box before marginalization
    newVol $\leftarrow \mathrm{vol}\,(x'_{\rho\vee})$ // Volume of box after marginalization
    $L_f \leftarrow L_f$ with each element f mapped to $f\left(\frac{\text{oldVol}}{\text{newVol}}\right)$
    newDens $\leftarrow \sum L_f$
    $(x'_{\rho\vee}, L_f) \leftarrow (x'_{\rho\vee}, \text{newDens})$
margLeaves $\leftarrow$ the set of leaves after marginalization
ancMap $\leftarrow$ Map from ancestor boxes in s' to their descendents in margLeaves
descDens $\leftarrow$ Map from each leaf in margLeaves to 0
for $(x'_{\rho\vee}, f'_{\rho\vee}) \in s'$ **do**
    if $x'_{\rho\vee} \in$ margLeaves **then**
        descDens$[x'_{\rho\vee}] \leftarrow$ descDens$[x'_{\rho\vee}] + f'_{\rho\vee}$
for $(x'_{\rho\vee} \mapsto L_d) \in$ ancMap **do** // L_d is list of descendents
    $L_d \leftarrow L_d \cap$ margLeaves
    for desc $\in L_d$ **do**
        descDens$[x'_{\rho\vee}] \leftarrow$ descDens$[x'_{\rho\vee}] + f'_{\rho\vee}$
$s' \leftarrow$ descDens as a list sorted according to the leaves
return s'

$(x_i)_{i\in\Lambda} = (x'_i)_{i\in\Lambda}$. This is non-normalized since in general $I^{|x'} = \int f^{|x'} d\left((x_i)_{i\in\Delta\setminus\Lambda}\right) \neq 1$. The conditional density is given by $\frac{f^{|x'}\left((x_i)_{i\in\Delta\setminus\Lambda}\right)}{I^{|x'}}$.

As with the previous operations, an algorithm for conditional density is presented in [7] and Algorithm 5 presented here is adapted for sparse bitstring representations of MRPs. In [7], this operation is called *slice* and for consistency that name is used here as well.

Algorithm 5: slice$(s, \Lambda, \boldsymbol{x}')$

Input: s, the $\mathbb{R}$-MRP ($e = 0$) representing the density-based histogram,
Λ, the set of axes to condition on,
$\boldsymbol{x}'$, the values to condition on at each axis in Λ
Output: s', the $\mathbb{R}$-MRP ($e = 0$) representing the histogram for the non-normalized conditional density

```
1 sliceBoxes ← set of boxes that contain x'
2 newBoxes ← sliceBoxes with coordinates in Λ removed and updated
    node labels
3 oldToNew ← Map from boxes in sliceBoxes to corresponding box in
    newBoxes
4 s' ← []
5 for (x_{ρ∨}, f_{ρ,∨}) ∈ s do
6     if x_{ρ∨} ∈ sliceBoxes then
7         s' ← s' :: (oldToNew[x_{ρ∨}], f_{ρ,∨})
8 return s'
```

3.4 Collated regular pavings (CRPs)

There are situations where several different histograms are needed at the same time, such as in density estimation where histograms are compared to determine which is the best estimate of the underlying density. For this reason, a data structure that contains several histograms can be advantageous.

One such data structure is presented in [13] as the *Collated Regular Paving* (CRP). A CRP is an RP where each leaf is associated with more than one value. In [13], the values associated with each leaf is the density of each histogram in the box that the leaf represents, as well as the empirical density of validation data in the box. Adding an additional histogram to the collection is referred to as *collating*, and the same term is used when a CRP is created from two non-collated histograms. To collate two MRPs, they must have the same root box, so that the leaf labels refer to the same sub-boxes.

Note that any two $\mathbb{A}-$MRPs with the same root box can be collated, this data structure is not unique to histograms. The only implementation presented

here is for density-based histograms, but the implementation details for other MRPs would be very similar.

A procedure for collation exists in `mrs2` [15], and the corresponding procedure for (sparse) bitstring representations of MRPs is presented here in Algorithm 6. Only the collation of two CRPs is presented since an MRP can easily be transformed into a CRP with only one underlying MRP. In this implementation, a CRP is an MRP where each leaf is associated with a map from $k \in \mathbb{K}$, where $\mathbb{K}$ is a set of labels (keys) for the histograms, to the density estimated by histogram f_k in the leaf's box. In other words, a CRP is a $\{\kappa : \mathbb{K} \to \mathbb{R}_{\geq 0}\}$-MRP. Every leaf has the same key set $\mathbb{K}$, so some leaves may have estimated densities of 0 for one or more histograms. This generally makes the CRP less sparse than the underlying histograms. If s is a CRP, let $K(s)$ denote the set of keys in s.

Algorithm 6: Collation of MRPs

Input: s_1, s_2, CRPs of density-based histograms with non-overlapping key sets

Output: s, the collation of s_1 and s_2

```
1  K_1 ← K(s_1); K_2 ← K(s_2)
2  Function collatorOp(m_1, m_2):
3      // m_1 and m_2 are maps from keys to densities, represented as sets {k ↦ v}
4      if m_1 = ∅ then // leaf does not exist in s_1
5          return m_2 ∪ {k ↦ 0 : k ∈ K_1}
6      else if m_2 = ∅ then // leaf does not exist in s_2
7          return m_1 ∪ {k ↦ 0 : k ∈ K_2}
8      else // leaf exists in both s_1 and s_2
9          return m_1 ∪ m_2
10 return MRPOperate(s_1, s_2, collatorOp, ∅)
```

Chapter 4

Scalable Histogram Estimation

Estimating histograms is one of the most natural approaches in density estimation. The main drawback of histograms is the computational resources needed when the number of dimensions and/or the sample size is very large.

A partial solution to this problem is to make the computation scalable, distributing the workload across several workers. In the best case scenario, such distributed computing decreases the computation time linearly in the number of workers.

The `mrs2` library for C++ [15, 7] contains efficient implementations for non-distributed estimation of histograms as MRPs. A distributed version implemented in Scala and using Apache Spark for distribution is given in [14]. In this work, the randomized algorithms for scalable density estimation based on minimum distance in [13] are made efficient with sparse MRPs.

4.1 Histogram estimation through merging

The algorithms presented in [7], [13] and [14] are top-down in the sense that they begin with a root box which is recursively split according to a randomized priority function. In [14] an algorithm for backtracking by merging leaves according to a priority function is presented, but only explored as far as backtracking after the splitting algorithm is run. Here, an alternative approach is presented, which is bottom-up, i.e. starting from each element in the dataset itself, and more naturally suitable to distributed computing.

4.1.1 Mapping to finest resolution

The bottom-up approach to MRP histogram estimation begins with determining a *finest resolution* from which the merging will start. Given a root box (which should be at least as large as the bounding box of the sample), the finest

resolution is determined by a depth D. The root box is then partitioned into an RP with each leaf at depth D, and each sample point is associated with the corresponding leaf box. D can be determined by, for example, the depth at which the volume of boxes is small enough, or all sidelengths are small enough. The finest resolution count-based histogram is given by counting the number of data points in each leaf box with depth D.

Since the bottom-up approach only merges leaves, this histogram should be much finer than the wanted final resolution, in terms of an optimally smoothed distribution of leaf depths $\leq D$. The finest resolution for a set of distinct sample data points should result in a single sample per leaf. Due to this extreme sparsity, empty leaves should not be stored, yielding a sparse $\mathbb{Z}-$MRP with neutral element 0. Once the finest resolution histogram is computed, the original sample is no longer needed, since the merges only depend on leaf counts.

This step is embarrasingly parallel even in a distributed setting since each sample point can be mapped to the appropriate leaf label independently, and the counting is a groupby and reduce operation. Map, groupby and reduce are basic operations in distributed computing.

This approach is given in Algorithm 7 and the output is not assumed to fit into memory of a single computer.

Algorithm 7: finestRes($x_{[n]}, \boldsymbol{x}_\rho, D$)

Input: $x_{[n]}$, the dataset. $\boldsymbol{x}_\rho$, the root box. D, the finest resolution depth.

Output: s, the $\mathbb{Z}-$MRP (e = 0) with counts at depth D.

```
foreach x ∈ x_[n] do // distributed map operation
    l ← leaf label of x at depth D
    x ← (x, l)
leafGroups ← x_[n] grouped by label // distributed grouping
foreach lp = (leaf, points) ∈ leafGroups do // distributed map
    lp ← (leaf, length(points))
s ← x_ρ // the trivial RP
𝕃(s) ← leafGroups
return s
```

4.1.2 Distributed backtracking

The next step from the extremely fine-resolution histogram is to backtrack until the number of leaves is small enough to fit into the available memory of a single computer. A non-distributed algorithm for this is presented in [14] as Algorithm 3. That algorithm takes a (sparse) histogram and a priority function $\varphi : \text{Count} \times \text{Volume} \to \mathbb{R}_{\geq 0}$ and returns a coarsening sequence of histograms, where each step in the sequence selects the parent node with lowest priority, and merges the children of that node. The priority function is randomized as

ties among several parent nodes with the lowest priority are resolved by uniform random sampling. Given a limit $\overline{\varphi}$ on the priority function, the merging process is stopped when the selected parent node has priority at least $\overline{\varphi}$. This ensures that all leaves have priority below $\overline{\varphi}$.

The distributed version of the algorithm takes advantage of the possibility of using sparse histograms by partitioning the histogram into subtrees that can be merged separately and then combined. This is done iteratively, where each step partitions the histogram into subtrees at a specific depth, and that depth decreases until no subtrees have mergeable nodes. The distributed algorithm is presented in Algorithm 8.

Algorithm 8: distBacktrack($\varphi, \overline{\varphi}, k, s$)

Input: $\varphi : \text{Count} \times \text{Volume} \to \mathbb{R}_{\geq 0}$, a priority function,
$\overline{\varphi}$, an upper limit on φ,
k, step size to decrease depth by each iteration,
s, the count-based histogram to be merged.
Output: s', count-based histogram where no more leaves can be merged.

```
D ← min(depth(s))
s' ← s
while D > 1 and s' has leaves that can be merged do
    if D ≤ k then
        k ← a new step size, smaller than D
    D ← D − k
    s_G ← s' grouped by ancestor at depth D // Distributed group by
    for s_i = (ancestor, L) ∈ s_G do // Distributed map, L is list of
     leaves descended from ancestor
        // Let x(l) be the box associated with leaf label l.
        if φ(Σ_{l∈L} #x(l), vol(ancestor)) < φ̄ then
            L ← [(ancestor, Σ_{l∈L} #x(l))]
        else
            L ← backtrack(φ, φ̄, L) // Non-distributed backtracking
              (Algorithm 3 from [14], stopping at φ̄)
    s_i ← L
return s
```

4.2 Minimum Distance Estimation

In [13], a method for obtaining a minimum distance estimate (MDE) histogram is presented. Such an MDE has universal performance guarantees [2]. This method builds on the `mrs2` package for C++ [15] and does not support parallel or distributed computations.

Let Θ be a finite set of indices for density estimates $\{f_{n,\theta} : \theta \in \Theta\}$. The number of bins in a sequence of coarsening histograms (represented as MRPs) constitutes such a set Θ. This Θ can be naturally ordered by the number of bins, and the goal is to select the optimal estimate from the $|\Theta|$ many histograms.

If $\nu \in (0, 1/2)$ is the fraction of data used for validation, then $n - \lfloor \nu n \rfloor$ (written $n - \nu n$ for convenience) points are used for training and the rest for validation.

For a pair $(\theta, \vartheta) \in \Theta^2$ where $\theta \neq \vartheta$, the *Scheffé set* of (θ, ϑ) is

$$A_{\theta,\vartheta} = A(f_{n-\nu n,\theta}, f_{n-\nu n,\vartheta}) = \{x : f_{n-\nu n,\theta}(x) > f_{n-\nu n,\vartheta}(x)\}.$$

The collection of all Scheffé sets over Θ is called the *Yatracos class* and is denoted $\mathcal{A}_\Theta$:

$$\mathcal{A}_\Theta = \{\{x : f_{n-\nu n,\theta}(x) > f_{n-\nu n,\vartheta}(x)\} : (\theta, \vartheta) \in \Theta^2, \theta \neq \vartheta\}.$$

The minimum distance estimate (MDE) $f_{n-\nu n,\theta^*}$ is the histogram constructed from training data of sample size $n - \nu n$ and validation data of size νn that minimizes

$$\Delta_\theta = \sup_{A \in \mathcal{A}_\Theta} \left| \int_A f_{n-\nu n,\theta} - \mu_{\nu n}(A) \right|$$

i.e. $\theta^* = \min_{\theta \in \Theta} \Delta_\theta$.

A procedure for finding this estimate is presented in [13] as a combination of Algorithms 2, 3, and 4. Since the size of the Yatrocos class grows as $|\Theta|^2$, it is infeasible to compare every estimate to every other estimate. To keep $|\Theta|$ fairly small, an adaptive search is employed, as done in [13].

A set of k estimates along the coarsening sequence are chosen equally far apart, and Θ is restricted to only those estimates. The best estimate $\hat{\theta}^*$ is chosen and $k - 1$ other estimates are chosen near $\hat{\theta}^*$, i.e. from 2 or more, but fewer than $k - 1$, nearest neighbors of $\hat{\theta}^*$. This process is repeated until no further "zooming in" is possible. The MDE is taken as the best estimate found. Usually, $10 \leq k \leq 20$.

Algorithms 9 and 10 presented here are adaptations of the algorithms in [13] to suit the sparse bitstring representation of MRPs. For the most part, these algorithms are not parallelized or distributed. The algorithms could possibly be further parallelized or distributed, but is not explored due to time constraints. However, the distributed backtracking procedure can be used to find the larget possible histogram that will fit in the memory of one machine, and MDE can be done from such a histogram, which is along the path of a consistent partitioning rule based on statistically equivalent blocks, due to the choice of the priority function used. Algorithms 2 to 10 are implemented in Scala and Apache Spark, and are published with a permissive open source license as [6].

Algorithm 9: getDelta(s, v)

Input: s, CRP of histograms to search,
v, count-based histogram ($\mathbb{Z}-$MRP) of validation data
Output: L, map from $\theta \in \Theta = \mathbb{K}(s)$ to Δ_θ

$\Theta \leftarrow \mathbb{K}(s)$
// compute $\mathcal{A}_\Theta$
$\mathcal{A}_\Theta \leftarrow []$
for $\theta \in \Theta$ **do**
  $r \leftarrow \{\vartheta \in \Theta : |\mathbb{L}(s_\theta)| < |\mathbb{L}(s_\vartheta)|\}$ // keys for finer histograms
  for $\vartheta \in r$ **do**
    $A_{\theta,\vartheta} \leftarrow \emptyset$
    for $(\boldsymbol{x}_{\rho\vee}, f_{\rho\vee}) \in \{(\boldsymbol{x}_{\rho\vee}, f_{\rho\vee}) \in s : f_{\rho\vee,\theta} > f_{\rho\vee,\vartheta}\}$ **do**
      $A_{\theta,\vartheta} \leftarrow A_{\theta,\vartheta} \cup \boldsymbol{x}_{\rho\vee}$
    $\mathcal{A}_\Theta \leftarrow A_{\theta,\vartheta} :: A$
make $\mathcal{A}_\Theta$ into a distributed array
$v \leftarrow \mathrm{CRP}(v, \theta_v)$ // CRP of v with key θ_v
$s \leftarrow \mathrm{collate}(s, v)$ // Add validation data to CRP
for $(\boldsymbol{x}, f) \in s$ **do** // compute measure for each key and subbox
  $f \leftarrow \{f_\theta \mathrm{vol}\,(\boldsymbol{x}) : f_\theta \in f\}$
for $A \in \mathcal{A}_\Theta$ **do** // distributed map
  $\mu_A \leftarrow \sum_{\boldsymbol{x}_{\rho\vee} \in A} f_{\rho\vee,\theta_v}$
  for $\theta \in \Theta$ **do**
    $\delta_{A,\theta} \leftarrow \left|\sum_{\boldsymbol{x}_{\rho\vee} \in A} f_{\rho\vee,\theta} - \mu_A\right|$
$L \leftarrow \{\theta \mapsto \max_{A \in \mathcal{A}_\Theta} \delta_{A,\theta} : \theta \in \Theta\}$
return L

Algorithm 10: mdeSearch(s, v, k, φ)

Input: s, fine count-based histogram as sparse MRP with root box $\boldsymbol{x}_\rho$,
v, validation data as distributed array,
k, step size in adaptive search,
φ, priority function for backtracking
Output: s', minimum distance estimate count-based histogram

Function mdeStep(s, v, k, $\overline{\varphi}$)**:**
  $B \leftarrow$ backtrack$(\varphi, \overline{\varphi}, s)$, // sequence of coarsening histograms (Algorithm 3 in [14])
  $L \leftarrow [s_{\theta_1}, \ldots, s_{\theta_k}]$, a list of equally spaced histograms in B
  $K \leftarrow \{\kappa_1, \ldots, \kappa_k\}$, CRP keys
  $s_c \leftarrow \mathrm{CRP}(s_{\theta_1}, \kappa_1)$
  for $s_{\theta_i} \in [s_{\theta_2}, \ldots, s_{\theta_k}]$ **do**
    $s_c \leftarrow \mathrm{collate}(s_c, \mathrm{CRP}(s_{\theta_i}, \kappa_i))$
  $s_v \leftarrow \mathrm{finestRes}(v, \boldsymbol{x}_\rho, \max(\mathrm{depth}(s)))$ // $\mathbb{Z}-$MDE of validation data
  $\Delta \leftarrow \mathrm{getDelta}(s_c, s_v)$
  $(\hat{\theta}^* \mapsto \hat{\delta}^*) \leftarrow \mathrm{argmin}_{(\theta \mapsto \delta) \in \Delta}\, \delta$ // Best histogram estimate found
  $i \leftarrow$ index of $\hat{\theta}^*$ such that $s_{\theta_i} = s_{\hat{\theta}^*}$
  return $(s_{\theta_{i-1}}, s_{\theta_i}, s_{\theta_{i+1}})$ // the best estimate and the two closest estimates
$(s'_+, s', s'_-) \leftarrow \mathrm{mdeStep}(s, v, k, 0)$ // backtracked to the trivial histogram with one leaf
while $\mathrm{size}(s'_+) - \mathrm{size}(s') > 1$ **do** // further zooming possible
  $\overline{\varphi} \leftarrow \varphi(s'_-)$
  $(s'_+, s', s'_-) \leftarrow \mathrm{mdeStep}(s'_+, v, k, \overline{\varphi})$ // start from finer estimate than best, limit backtracking to coarser estimate than best
return s'

distribution	dimension	n	$\overline{\varphi}$	L_1 error	time (h:m:s)
Uniform	1	10^7	1000	$1.13 \cdot 10^{-3}$	00:08:14
	10	10^7	1000	$6.07 \cdot 10^{-4}$	00:09:41
	100	10^7	1000	0^*	00:16:41
	1000	10^7	1000	$3.44 \cdot 10^{-3}$	01:41:56
Uniform	1	10^8	10000	$1.13 \cdot 10^{-3}$	00:26:14
	10	10^8	10000	0^*	00:31:57
	100	10^8	10000	0^*	01:19:15
	1000	10^8	10000	0^*	10:04:55
Gaussian	1	10^7	1000	$4.63 \cdot 10^{-3}$	00:13:23
	2	10^7	1000	N/A	00:22:37
	5	10^7	1000	N/A	00:37:49
Gaussian	1	10^8	10000	$3.50 \cdot 10^{-3}$	00:17:48
	2	10^8	10000	N/A	00:39:08
	5	10^8	10000	N/A	01:22:56

Table 4.1: Performance of MDE histogram estimation. Histograms with errors marked with * have only one leaf (leading to 0 error for Uniform distributions)

4.3 Results

To assess the performance of the histogram estimation algorithms, two sets of simulated test cases are investigated. The first test case consists of Uniform distributions over the unit hypercube in 1, 10, 100, and 1000 dimensions, and the second test case consists of standard Gaussian distributions in 1, 2, and 5 dimensions.

The same priority function $\varphi(\text{count}, \text{volume}) = \text{count}$ is used for both test cases, both in Algorithms 8 and 10. This priority function merges leaves with a low count first, and corresponds to `SEBTreeMC` in [13]. The limit $\overline{\varphi}$ in Algorithm 8 is set so that Algorithm 10 does not exceed the memory limit of one machine.

Each test case is simulated using 10^7 and 10^8 training data points, and half as many validation data points. The reported timings include simulation of the data set.

The simulations for Uniform distributions use a cluster with four workers and a driver with 4 cores and 8 GB memory each, for a total distributed performance of 16 cores and 32 GB memory and a non-distributed performance of 4 cores and 8 GB memory.

The simulations for Gaussian distributions use a cluster consisting of nine workers and a driver with 2 cores and 8 GB memory each, for a total distributed performance of 18 cores and 72 GB memory and a non-distributed performance of 2 cores and 8 GB memory.

L1 errors are not computed for multivariate Gaussian distributions due to the difficulty of numerically integrating the density to a sufficient tolerance. To determine the estimate's usefulness as a discriminator between different densities, two Gaussian distributions with different means are simulated and used to

2D Multivariate Gaussian			5D Multivariate Gaussian		
$\delta\sqrt{2}$	confusion matrix (%)		$\delta\sqrt{5}$	confusion matrix (%)	
1	TP: 34.4	FP: 15.6	1	TP: 34.3	FP: 15.7
	FN: 15.4	TN: 34.6		FN: 16.0	TN: 34.0
2	TP: 41.9	FP: 8.08	2	TP: 41.7	FP: 8.28
	FN: 8.05	TN: 41.9		FN: 8.36	TN: 41.6
3	TP: 46.5	FP: 3.52	3	TP: 46.2	FP: 3.78
	FN: 3.49	TN: 46.5		FN: 3.73	TN: 46.3
5	TP: 49.7	FP: 0.26	5	TP: 49.5	FP: 0.50
	FN: 0.29	TN: 49.7		FN: 0.43	TN: 49.6

Table 4.2: Confusion matrices for discrimination based on MDE histograms between pairs of Gaussian distributions. Since an equal number of points are tested from both distributions, the perfect discriminator has TP and TN at 50%, and FP and FN at 0%.

estimate two histograms. New sample points are then generated and assigned to the histogram with the highest estimated density at that point. Repeating this for multiple pairs of Gaussian densities with varying similarity, the discriminatory property of the histograms can be observed.

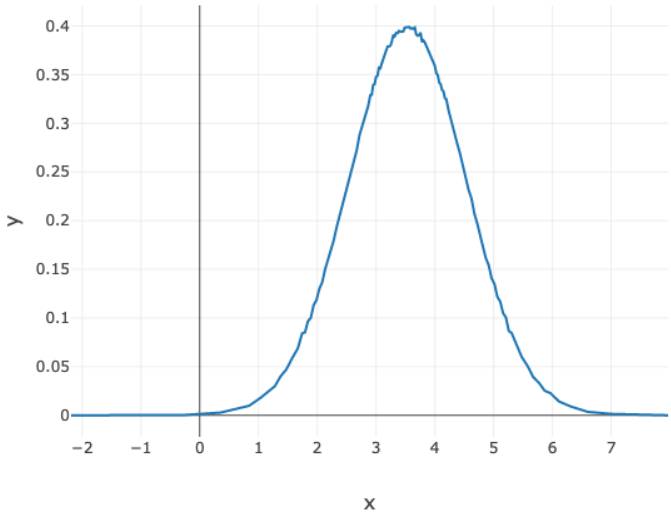
Four such pairs are constructed using an equal number of training and validation points ($5 \cdot 10^6$ training and $2.5 \cdot 10^6$ validation). The "base" distribution is the same in all pairs, the standard multivariate Gaussian distribution. The other distributions are standard multivariate Gaussians shifted by a vector $\delta\mathbf{1}$, where $\mathbf{1}$ is the vector with 1 in every component. The values for δ are chosen such that there are integer Euclidean distances between the distributions' means. Finally, 10^4 new sample points are generated from each distribution and used to obtain the confusion matrices. The results are presented in Table 4.2.

4.4 Discussion

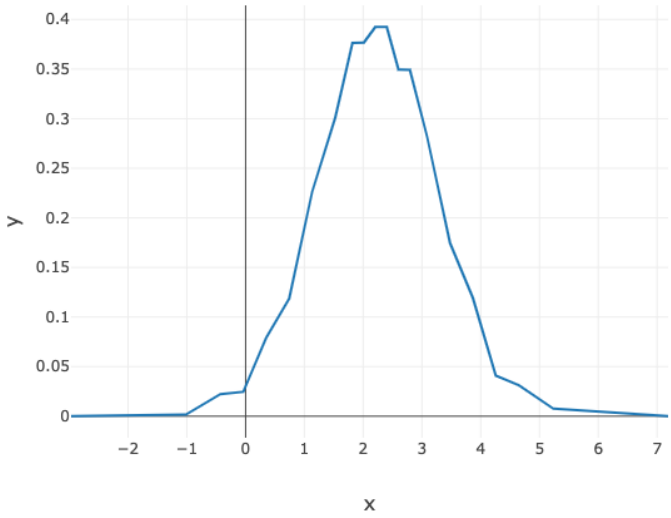
As seen in tables 4.1 and 4.2, the performance of the MDE histograms is promising. The L_1 errors decrease as the sample size increases, although a method to compute the L_1 error for systematic approximations to multivariate Gaussian distributions would be useful to establish this further. Such a method is presented in [13] by creating an MRP of the underlying density and sampling from that. Due to time constraints this is not implemented in this project, but it would be a natural extension to further investigate the usefulness of the algorithms and procedures presented here.

Figure 4.1 shows the marginalization of 2D and 5D Multivariate Gaussian histograms. The results are as expected, so together with the results in Tables 4.1 and 4.2, the histogram estimation seems to be quite performant.

The performance of the MDE histograms could possibly be increased by tuning the backtracking priority function φ and its limit $\overline{\varphi}$. In the simulations



(a) 2D, $\delta = 5/\sqrt{2} \approx 3.5$



(b) 5D, $\delta = 5/\sqrt{5} \approx 2.2$

Figure 4.1: 1D Marginalizations of histograms estimated from Multivariate Gaussian samples with mean $\delta\mathbf{1}$ and identity covariance matrix. The marginalized histogram based on a bivariate Gaussian is much smoother, which is due to there being more leaves per dimension in that histogram before marginalization.

presented here no particular care was taken to tune these parameters further than to the point where the procedure stays within the memory limits of the workers.

All distributed algorithms in this project are implemented without regard for how the data is partitioned, which can increase the level of necessary communication between workers. Such communication is generally the most severe bottleneck in distributed systems and should be limited to the extent possible. Hence, the procedures as implemented should be seen as a worst-case analysis corresponding to the scenario where there is no control over the way data is stored. Such a situation can arise if a data set is collected and stored without a specific analysis in mind, and when the analysis begins the data set is too large to repartition without spending an unreasonable amount of time or computing power.

In the case of Algorithms 7 and 8, a natural partition scheme is given by sorting leaves using the ordering defined in Section 3.1.1. In Apache Spark, this is available as `repartitionByRange`. This can result in a distributed MRP where each partition contains a subtree, at which point no communication between workers is necessary until the final result is collected. Distributed `groupBy` and `mapGroups` operations are thus replaced with `mapPartition` operations, which does not result in inter-worker communication. Implementing this would be fairly straightforward, but left out due to not being an option in the worst-case scenario described above.

Support

This project was partly supported by *Databricks University Alliance* with infrastructure credits from AWS *AWS* to Raazesh Sainusiin, Department of Mathematics, Uppsala University, Sweden, and by Combient Competence Centre for Data Engineering Sciences, Department of Mathematics, Uppsala University, Sweden.

Bibliography

[1] Logical Clocks. Maggy, a framework for distribution transparent machine learning experiments on apache spark. `https://maggy.ai`.

[2] Luc Devroye and Gabor Lugosi. *Combinatorial Methods in Density Estimation*. Springer New York, NY, 2001.

[3] R. Hammer et. al. *C++ toolbox for verified computing: theory, algorithms, and programs*. Springer-Verlag, Berlin Heidelberg, 1995.

[4] Johannes Graner. Spark trend calculus. `https://github.com/lamastex/spark-trend-calculus`, 2020.

[5] Johannes Graner, Albert Nilsson, and Raazesh Sainudiin. Spark trend calculus examples. `https://lamastex.github.io/spark-trend-calculus-examples/`, 2020.

[6] Johannes Graner and Tilo Wiklund. Sparkdensitytree, a scala and apache spark library for distributed histogram estimation. `https://github.com/johannes-graner/SparkDensityTree`, 2022.

[7] Jennifer Harlow, Raazesh Sainudiin, and Warwick Tucker. Mapped regular pavings. *Reliable Computing*, 16:252–282, 2012.

[8] HistData. Free forex historical data. `https://histdata.com`, 2022.

[9] Andrew Morgan. TrendCalculus: A data science for studying trends. `http://bytesumo.com/blog/2015/01/trendcalculus-data-science-studying-trends`, 2015.

[10] Andrew Morgan, Antoine Amend, David George, and Matthew Hallett. *Mastering Spark for Data Science*. Packt Publishing, 2017.

[11] Andrew Nobel and Gabor Lugosi. Consistency of data-driven histogram methods for density estimation and classification. *The Annals of Statistics*, 24:687–706, 1996.

[12] Raazesh Sainudiin. Enclosing the maximum likelihood of the simplest dna model evolving on fixed topologies: Towards a rigorous framework for phylogenetic inference. Technical report, Cornell University, Ithaca, USA, 2004.

[13] Raazesh Sainudiin and Gloria Teng. Minimum distance estimation with universal performance guarantees over statistical regular pavings. *Japanese Journal of Statistics and Data Science*, 2019.

[14] Raazesh Sainudiin, Warwick Tucker, and Tilo Wiklund. Scalable multivariate histograms. `http://lamastex.org/preprints/20180506_SparkDensityTree.pdf`, 2020.

[15] Raazesh Sainudiin, Thomas York, Jennifer Harlow, Gloria Teng, Warwick Tucker, and Dillon George. MRS 2.0, a C++ class library for statistical set processing and computer-aided proofs in statistics. `https://github.com/lamastex/mrs2`.