University of Western Australia

Honours project

Gradient Boosting Decision Tree Training with Swarm Intelligence

Xi Rao

School of Physics, Mathematics and Computing

22435044@student.uwa.edu.au

Supervised by

Dr Zeyi Wen
School of Physics, Mathematics and Computing

23 May 2022

Gradient Boosting Decision Tree Training with Swarm Intelligence

by

XI RAO

This dissertation is submitted as partial fulfilment of

the requirements for the

Honours Programme

in

Computer Science and Software Engineer

in the

School of Physics, Mathematics and Computing

in the

The University of Western Australia

Semester 1, 2022

Acknowledgments

I want to thank Dr Zeyi Wen for being my supervisor and offering me an endless source of help for this work. He has always been my mentor and guided me through several research projects during my undergraduate study. I want to thank Mr Hanfeng Liu for providing me with the original inspiration for the work and for his precious advice to help me solve the technical challenges. I want to thank Mr Jiacheng Qiu and Miss Yuxi Lu for providing me with the feedback to improve the thesis presentation.

Finally, I want to thank my family and friends for showing support and encouragement throughout my four years of undergraduate study at UWA.

Contents

List of Figures

List of Tables

Abstract

The Gradient Boosting Decision Tree (GBDT) attracts significant attention in machine learning with its state-of-the-art performance in solving real-world problems. Existing cutting-edge GBDT implementations such as XGboost and LightGBM rely on greedy algorithms to determine the best split point for every internal node. Specifically, for each internal node, the greedy algorithm needs to scan every data instance for each feature to evaluate the gain of all possible spit points. Such methods are usually imprisoned in a local optimum solution and are hard to optimise.

Swarm Intelligence has been widely applied for various optimisation tasks with its good global search capability. In this dissertation, we propose several swarm-based approaches to replace the greedy algorithms in GBDT training. In recent years, Swarm Intelligence has been successfully employed for hyper-parameters tuning to improve the predictive performance in the GBDT training. However, to the best of our knowledge, swarm-based GBDT induction remains an unexplored research area. Our swarm-based techniques can build a GBDT solution using the split points that are explored by the swarm. As such, they may find a global optimal solution. Evaluation on the seven commonly-used datasets shows that our swarm-based approaches outperform existing state-of-the-art GBDT implementations.

1 Introduction

The Gradient Boosting Decision Tree (GBDT) has been one of the most popular machine learning algorithms recently due to its capacity to build state-of-the-art data science solutions. It not only shows outstanding performance in the Kaggle data mining competition in recent years but also is widely used in various real-world applications, such as intrusion detection systems [1], travel time forecasting [2] and web search ranking [3]. Inspired by the ensemble learning method, a GBDT model is built by constructing a series of decision trees. The decision trees are built using a greedy algorithm to find the best split for every internal node. However, Laurent and Rivest [4] previously demonstrated that creating an optimal decision tree is NP-Complete. Current greedy GBDT methods rely on the hand-tuned heuristic to build the decision trees. Examples include minimising the split Mean Square Error (MSE) [5] and maximising the split second-order gain [6]. Such methods do not guarantee to induce an optimal GBDT solution, and it is likely to reach a local optimum. Swarm intelligence algorithms have been researched to build a single decision tree over the last two decades [7]–[11], which shows it may be a better way to obtain an optimal solution for the decision tree induction. We can thus extend these ideas to train GBDTs.

Swarm Intelligence is a well-known branch of computational intelligence used to deal with complex optimisation problems. Some well-known swarm intelligence algorithms, including Artificial Bee Colony (ABC) [12], Bat algorithm (BA)[13], Particle Swarm Optimization (PSO) [14], and Ant Colony Optimization (ACO) [15], are inspired by the behaviour of social animals in nature, such as bird swarm. PSO is one of the most common swarm intelligence algorithms, widely employed in various optimisation tasks due to its ability to tackle non-convex problems with ease and efficacy. PSO is inspired by simulating the predation behaviour of birds. The core idea of the PSO algorithm is to leverage information exchange among individuals in the group such that the movement of the entire population in the problem-solving space evolves from irregular to regular. Compared to other swarm intelligence algorithms, PSO has the following advantages; (i) PSO is easy to implement; (ii) PSO has fewer parameters to adjust. (iii) PSO is highly scalable because it is easy to speed up by parallel computing [16].

In this research project, we propose two novel approaches that use the PSO algorithm to train a GBDT model instead of using greedy algorithms. Our methods for constructing decision trees do not rely on hand-tuned heuristics to determine the best split point at each internal node; instead, we employ a collection of supplied split points that are explored by PSO. Such a global search strategy may potentially find a better tree structure than a greedy method. In summary, we specify our major contributions to this dissertation as follows.

1. We propose Swarm Gradient Boosting Decision Tree (SGBDT) to train a GBDT model using the PSO algorithm. SGBDT initialise the swarm by random feature selections and force the global search to find the optimal solution.

2. We apply the feature discretisation algorithm to reduce the search space of SGBDT to improve the quality of global search.

3. We propose a Pre-trained Swarm Gradient Boosting Decision Tree (PSGBDT) to further improve SGBDT by initialising the swarm with the pre-trained results from the state-of-the-art GBDT implementations.

4. We conduct extensive experiments on seven commonly-used datasets and show that SGBDT can provide comparable results to state-of-the-art GBDT implementations, and PSGBDT outperforms state-of-the-art GBDT implementations.

The rest of this dissertation is organised as follows. First, we provide preliminaries to help understand the GBDT and PSO algorithms in Section 2. Then, we review the related works in the field of GBDT and PSO in Section 3. We follow by concluding the literature review and specify the objectives of this research project in Section 4. After that, we explain the relevant details of our proposed approaches in Section 5. Section 6 identifies multiple objectives for the experiments. Section 7 presents and discusses the experimental results. Section 8 outlines the limitations of our proposed approaches and introduces future works. Section 9 makes a conclusion for the dissertation.

2 Preliminaries

In this section, we briefly introduce decision tree and ensemble learning. Then, we discuss the key ideas and formulations of GBDTs and PSO, so as to have some necessary technical background in understanding the related works and methodologies presented in the following sections.

2.1 Decision Trees

A decision tree construction can be seen as the process of continuous feature selection. A sub-optimal decision tree can be constructed by step-wise splitting. Each node computes the information contribution of each feature and chooses the one that has the most significant contribution as the split point. Finally, the constructed decision tree includes a set of if-then rules, and each rule specifies a path from the root node to a leave, where each leave represents the result of each rule. Decision tree learning is to learn a set of if-then rules that can minimise the training errors and maximise the generalisation ability. Hence, the main difference between modern decision tree algorithms is how to define heuristic learning rules and improve the generalisation ability of decision trees.

Breiman et al. introduced Classification and Regression Trees (CART) in 1984 [5]. CART is capable of dealing with both classification and regression tasks. It uses the Gini index and MSE as the split metric to find the optimal split point for the classification and regression tasks. CART, in other words, enumerates all features for each internal node and computes the MSE or Gini index. And then chooses the split point with the minimum MSE or maximum Gini index to be partitioned.

Quinlan proposed the ID3 [17] in 1996 and uses the entropy as the matric to evaluate the split point. For example, ID3 selects the feature and threshold that either minimises entropy or maximises information gain for each internal node. However, ID3 is constrained by being incapable of processing the dataset that contains the categorical features. To overcome the problem, Quinlan's subsequent works proposed the C4.5 algorithm [17]. C4.5 improves the ID3 algorithm in terms of data processing, accuracy, generation ability, and inference speed. The following points specifies the enhancements of C4.5.

1. In terms of data processing, C4.5 can handle both continuous and discrete datasets.
2. C4.5 evaluates the splitting features using the gain ratio impurity and selects the one with the highest score. Because information gain cannot distinguish between features with different amounts of thresholds but the same information gain score, this measure may enhance classification accuracy.
3. C4.5 prunes the tree once it has been built. It enables the decision tree to trim branches with little impact on the classification outcome, lowering the risk of overfitting. Furthermore, by reducing the size of the tree, inference time can be decreased.

C4.5 has become one of the most popular decision tree induction methods with these considerable enhancements. Numerous research indicates that C4.5 has outstanding prediction accuracy and inference performance among decision tree approaches for the classification tasks [18].

2.2 Ensemble Learning

The goal of ensemble learning is to combine multiple weak prediction models to create a more resilient prediction model. The main advantage of this learning method is that even if one of the weak prediction models predicts an incorrect result, the error can be corrected by other weak prediction models. As a result of this advantage, ensemble learning methods perform outstandingly for a wide range of machine learning tasks. Numerous researchers have demonstrated its good performance through multiple experiments [19]–[21].

The two most commonly used ensemble learning methods are bootstrap aggregation (or bagging) [22] and boosting [19]. Bagging employs the bootstrap method to sample N datasets from the entire dataset (with replacement). Then, it trains the model on each sub-dataset to make a prediction and combine the predicted results of the N models to obtain the final result. Specifically, for the classification problem, bagging votes on N prediction models to obtain the final result. For the regression problem, bagging obtains the eventual result by calculating the average value for the N prediction models. Moreover, bagging is an efficient method due to its high parallelizability - each model is trained independently on a sub-dataset; thus, we can apply multi-thread programming to reduce the training time. Boosting, on the other hand, is seen to be processed consecutively because the output of one model is the input of the next. The aim behind boosting is to fit the data using a weak prediction model initially. Then, the process constructs the next weak prediction model to minimise the weakness of the preceding model until a more robust one is generated. Gradient boosting, which is utilised in GBDTs, is a variant method of boosting.

Gradient boosting is proposed by Friedman [23] to address the machine learning problems such as classification and regression. Its idea comes from gradient descent. Nevertheless, different from the gradient descent, gradient boosting iteratively creates a series of weak prediction models. The newly added weak prediction model trained in each iteration is based on the negative gradient [24] information of the previous model's loss function. Eventually, the predicted result of all the weak prediction models is aggregated to produce the final result. Compared to gradient descent, both methods use the negative gradient information to update the current model. However, the model is expressed in the form of parameters in gradient descent, so the gradient is descended by updating parameters. In gradient boosting, the descent of the gradient is obtained through the summation of the models. The GBDTs is a method of using gradient boosting. In this situation, the individual decision tree represents the weak prediction model.

2.3 Gradient Boosting Decision Trees

Standard GBDTs training applies CART regression tree as the weak learner. Hence, the process of generating a decision tree calculates the MSE for every internal nodes in order to find the best split point. The exact process is carried out recursively until the entire tree is generated. Then, we can generate the next decision tree based on the negative gradient information of the loss function to reduce the residual. The exact process iteratively proceeds until the training is completed. Thus, we have completed the training of a standard GBDTs model.

We explain the complete process of standard GBDTs training in the following text.

Given a dataset $\{(x_i, y_i)\}_{i=1}^n$, where x_i represents a sample,and y_i is the label of the sample x_i. Assume $F(x)$ represents the current prediction model, and $L(y, F(x))$ represents the loss function. For any x_i that belongs to the dataset $\{(x_i, y_i)\}_{i=1}^n$,the current prediction is $F(x_i)$, and $L(y_i, F(x_i))$ is loss between the label y_i and the current prediction $F(x_i)$. The GBDT algorithm aims to learn an optimal prediction model $F(x)$ that can minimise the integrated loss function $\sum_{i=0}^n L(y, F(x))$.

In order to find an optimal prediction model $F(x)$, the GBDT first creates an initial decision tree $F_0(x)$, then iteratively builds M new decision trees. In each iteration, the GBDT adds a new decision tree $h_m(x)$, which is trained base on the loss function $L(y, F(x))$ to reduce the residual. In other words, the new decision tree $h_m(x)$ is trained to predict the residual rather than the actual value. Eventually, the final prediction model can be defined as follows.

$$F_m(x) = F_0(x) + \sum_{m=1}^{M} \upsilon \lambda_m h_m(x) \tag{1}$$

where the shrinkage parameter υ is used to control the learning rate, and gamma is the weight of $h_m(x)$. Each base learner multiplies a shrinkage parameter υ to limit its contribution to the final model, which can reduce the probability of overfitting. The training process of the GBDT can be summarised as 3 steps, which are presented as follows.

Step 1: Initialise the first model $F_0(x)$ with a constant value, where N is the number of data instances.

$$F_0(x) = argmin_\lambda \sum_{i=1}^{N} L(y, \lambda) \tag{2}$$

Step 2: Building M decision trees, m = 1,2,3,....,M:

a) For each iteration, we calculate the negative gradient of the residual r_{im}, where r_{im} represents the residual (or pseudo residual).

$$r_{im} = -[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}]_{F(x)=F_{m-1}(x)} \qquad i = [0, 1, 2, ..., N] \tag{3}$$

b) Train a new decision tree $h_m(x_i)$ to fit the residual base on the training dataset $\{(x_i, r_{im})\}_{i=1}^{n}$. Let J_m to represent the number of leaves of the *m-th* decision tree. And R_{jm} represents the terminal region.

$$h_m(x) = \sum_{j=1}^{J_m} b_{jm} \mathbb{1}_{R_{jm}}(x) \tag{4}$$

where b_{jm} stands for the predicted value in the region R_{jm}.

c) Calculate the leave's weights of $h_m(x_i)$ by solving the following one-dimensional optimisation problem, to minimise the loss.

$$\gamma_m = argmin_{\gamma} \sum_{i=1}^{N} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)) \tag{5}$$

d) Update the model as follows:

$$F_m(x) = F_{m-1}(x) + \upsilon \gamma_m h_m(x) \tag{6}$$

Step 3: Using $F_M(x)$ as the final prediction model.

The GBDT performs outstandingly in classification and regression problems. However, the GBDT has a little variation in solving these two problems. Friedman [25] proposed several loss functions for GBDTs. For regression problems, GBDTs can apply the least square function, the least absolute deviation function and the Huber function as the loss function. For classification problems, due to the sample output being a discrete value rather than a continuous value, we cannot directly fit the residual from the categorical result. The solution is to use the regression tree that we utilise for the regression problem, but with a loss function of negative log-likelihood [25].

We specify how to calculate the leave weights of the trees using various loss functions in the following text. With leave weights, the model updating can be quickly addressed by equation 6. We omit detail of mathematical derivations because they can be found in [25].

Regression

For a regression problem, we use least square as loss function.

$$L(y_i, F(x_i)) = \frac{(y_i - F(x_i))^2}{2} \tag{7}$$

The leave weight γ_{jm} is formed as

$$\gamma_{jm} = argmin_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F(x_i) + \gamma) \tag{8}$$

Replacing $L(y_i, F(x_i))$ to Equation (7), we get:

$$\gamma_{jm} = average_{x_i \in R_{jm}} r_{im} \tag{9}$$

Hence, we minimise the loss by obtaining the average of r_{im}, as shown in Equation (9).

Classification

For a classification problem, we use negative log-likelihood as loss function. The negative log-likelihood is written as

$$L(y_i, F(x_i)) = -y_i log F(x_i) - (1 - y_i) log(1 - F(x_i)) \tag{10}$$

We have sigmoid function

$$F(x_i) = \frac{1}{1 + e^{-F(x_i)}} \tag{11}$$

Replacing $F(x_i)$ to sigmoid function, we have

$$L(y_i, F(x_i)) = y_i log(1 + e^{-F(x_i)}) + (1 - y_i)[F(x_i) + log(1 + e^{-F(x_i)})] \tag{12}$$

The leave weight γ_{jm} is formed as:

$$\gamma_{jm} = argmin_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F(x_i) + \gamma) \tag{13}$$

Replacing $L(y_i, F(x_i))$ to Equation (13), we get

$$\gamma_{jm} = \frac{\sum_{x_i \in R_{jm}} r_{im}}{\sum_{x_i \in R_{jm}} (y_i - r_{im})(1 - y_i + r_{im})} \tag{14}$$

where the γ_{jm} is approximated using Newton-Raphson [25]. Hence, we minimise the loss using Equation (14)

Example

Suppose we have a regression task to predict the housing price. The training set (Table 1) include two features, industry numbers (Ind) and populations (Pop).

Table 1: Example GBDT training dataset

	Ind	Pop	Label
0	100	10k	15k
1	200	20k	90k
2	600	58k	110k
3	80	61k	180k
4	1200	5k	580k
5	1011	250k	780k

First, we initialise the model for the training set using Equation 2.

$$\sum_{i=1}^{N} \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = \sum_{i=1}^{N} \frac{\partial(\frac{1}{2}(y_i - F(x_i))^2}{\partial F(x_i)} = \sum_{i=1}^{N} F(x_i) - y_i$$

Make the derivative to 0, we get

$$F(x_i) = \frac{\sum_{i=1}^{N} y_i}{N}$$

So the initial model $F_0(x)$ is the average of sample labels, thus $F_0(x) = (15 + 90 + 110 + 180 + 580 + 780) / 6 = 292.5$k. Then, we can calculate the residuals between labels and $F_0(x)$ using the Equation (3). Noting the computed residuals as $r_{1,1}$.

Table 2: This table present the results of $F_0(x)$ calculations and negative gradient calculations

	Ind	Pop	Label	$F_0(x)$	$r_{1,1}$
0	100	10k	15k	292.5k	-277.5k
1	200	20k	90k	292.5k	-202.5k
2	600	58k	110k	292.5k	-182.5k
3	80	61k	180k	292.5k	-112.5k
4	1200	5k	580k	292.5k	287.5k
5	1011	250k	780k	292.5k	487.5k

Using $r_{1,1}$ as the targets to fit a CART tree $h_1(x)$. Suppose $h_1(x)$ has only two leaves.

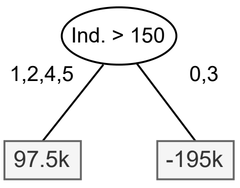


Figure 1: An Example CART tree $h1(x)$. [1,2,4,5] and [0,3] stands for the remain indices for the left and right split

According to Equation 9, we can calculate the weight of leaves $\gamma_{1,1}$, $\gamma_{1,2}$ as follows.

$$x_i \in R_{1,1}, \qquad \gamma_{1,1} = \frac{(-202.5 - 182.5 + 287.5 + 487.5)}{4} = 97.5k$$

$$x_i \in R_{1,2}, \qquad \gamma_{1,2} = \frac{-(277.5 + 112.5)}{2} = -195k$$

Once we finish calculating the leave weights, we can update the model using Equation (6). After that, GBDT backs to calculate the residuals between $F_0(x)$ and $r_{1,1}$ to get $r_{1,2}$, and using $r_{1,2}$ as targets to fit the second CART tree $h_2(x)$. The process will be continue iterating until $m = M$.

2.4 Particle Swarm Optimisation

The Particle Swarm Optimisation (PSO) algorithm is a global optimisation algorithm that originated from the study of swarm behaviour in nature (e.g. bird colonies). It can be used to solve various optimisation tasks by simulating social behaviour [1]. Specifically, we can define a group of particles, and the goal of PSO is to use the share information of each particle in a group so that the entire group can search for the global optimal solution in the problem-solving space.

In the PSO algorithm, we define L as the population and use $\{s_i\}_{i=1}^{L}$ to represent a group of particles (or candidate solutions). For each particle s_i , it can be represented as a point in N-dimensional space, where N stands for the total dimensions of an optimisation task. Each particle maintains its own $v_i(v_i \in R^N)$, and $p_i(p_i \in R^N)$, where v_i represents the velocity of the *i-th* particle, p_i stands for the position of the i^{th} particle. Also, each particle in the swarm is able to access two pieces of information, which are $pbest_i$ and $gbest$. $pbest_i$ represents the best-evaluated position that the *i-th* particle has been searched. $gbest$ represents the best-found position searched by the whole swarm so far. Then, we can update the velocity and position of the current particle s_i according to $pbest_i$ and $gbest$. Once we update the velocity and position of the particles, each particle can move to the new position with the updated velocity.

Formally, PSO utilises two main formulas for updating. One is for updating the *d-th* $(d \in N)$ dimension velocity of the *i-th* particle, and the equation is given below.

$$v_{(i,d)} = \omega v_{i,d} + c1r1(pbest_{i,d} - p_{i,d}) + c2r2(gbest_d - p_{i,d}) \tag{15}$$

where ω represents the inertia of a particle, $c1$ and $c2$ are the cognitive learning factor, and $r1$ and $r2$ are the random weight. Inertia can be explained by how much its previous velocity influence its next velocity. The inertia version of PSO

[26] was proposed soon after the initial version of PSO [14]. The cognitive learning factor $c1$ determines the influence of the locally best-found position in history, and $c2$ determines the influence of the best-found position of the whole swarm. $r1$ and $r2$ are the random weights in $[0, 1]$, which are used to enhance the randomness of search.

Once the velocity is updated, the position of the current particle can be computed by another formula shown as follows.

$$p'_{i,d} = p_{i,d} + v_{i,d} \tag{16}$$

The concrete process for the PSO algorithm is demonstrated in Algorithm 1 below. As you can see from the pseudocode, we can summarise the whole search process into three main steps. In Step 1 (from lines 1-7), we randomly initialise the velocity, position and locally best-found position for each particle s_i. Then, the particle s_i can be represented as $s_i = (v_i, p_i, pbest_i)$.In Step 2 from lines 10 - 11, we update the velocity and position for each particle by using Equation(15) and Equation(16) that we have elaborated earlier. In the last step, we evaluate each particle by the function f, where f is the function to produce fitness value for each particle. After evaluation, we update $pbest_i$ and $gbest$.

To summarise, the PSO algorithm can be seen as a search process to obtain an optimal solution for an optimisation task. This process is achieved by a swarm, each member in the swarm called a particle, which is also defined as a potential solution for an optimisation task in the N-dimensional space. Each particle is able to memorise the best-found position of the swarm ($gbest$), and the best-found position in its own memory $pbest_i$. In each iteration, every particle in the swarm shares its information to adjust the velocity of each dimension to calculate the new position of the particle. Particles constantly search and change their states in the multi-dimensional space until they reach an equilibrium state or exceed the computational limit.

Algorithm 1: An algorithm with PSO

```
Input : Number of iterations $I$
        The size populations $L$
        Dimensions of the optimisation problem $N$
        Inertia weight $\omega$
        global search weight $c1$
        local search wright $c2$
1  for $i \leftarrow 0$ to $M$ do
2      for $j \leftarrow 0$ to $N$ do
3          $v_{i,d} \leftarrow$ random velocity;
4          $p_{i,d} \leftarrow$ random position;
5          $pbest_{i,d} \leftarrow p_{i,d}$;
6      end
7  end
8  for $t \leftarrow 0$ to $I$ do
9      for $i \leftarrow 0$ to $L$ do
10         update $v_i$ by equation (15)
11         update $p_i$ by equation (16)
12         if $f(p_i) < f(gbest)$ then
13             update $gbest$;
14         end
15         if $f(p_i) < f(pbest_i)$ then
16             update $pbest_i$;
17         end
18     end
19 end
```

3 Related Works

In this section, we discuss the related works of GBDTs and swarm-based decision tree training. We first present the mainstream libraries and their key principles for training GBDTs. Then, we describe the key related works on using PSO for training decision trees.

3.1 GBDTs and GBDT Training Systems

Standard GBDT training forms the foundations for the mainstream GBDT implementations. Based on the previous works, we explain how to train the GBDT using two state-of-the-art implementations, XGBoost and LightGBM.

Training GBDTs with XGBoost

XGBoost [6] is a method of extreme gradient boosting, which is a specific implementation of GBDT. However, compared to the classic GBDT model, XGBoost has done a lot of optimisations to improve accuracy and efficiency. The boosting strategy of XGBoost is similar to GBDTs, both aiming at the newly added learner to further reduce the difference between the predicted value and the real value (residual). However, instead of just computing the gradient information of the loss function, XGBoost performs a second-order Taylor expansion on the customised objective function [6]. Compared with the classic GBDT that applies the least square function as a loss function, this method is more accurate and efficient [6].

In XGBoost, the customised objective function can be represented as follows.

$$Obj = \sum_i L(y, F(x)) + \sum_k \Omega(f_k) \tag{17}$$

The objective function is defined by two parts. The first part $L(y, F(x))$ stands for the loss function, which can be any convex differentiable loss function. $\Omega(f_k)$ in the second part represents the regularisation term to control the complexity of the trees to reduce the probability of overfitting. This is one of the significant improvements that make XGBoost better than the classic GBDT.

Moreover, XGBoost has customised its own rule of computing the gain, and this self-defined gain rule is derived from the second-order Taylor expansion of the customised objective function. After the second-order Taylor expansion for the objective function, we can get the information of gradients and second-order derivatives. Let G denotes the gradients, and H stands for the second-order derivatives. We can therefore obtain the customised gain rules as follows.

$$Gain_{split} = \frac{1}{2}[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} + \frac{G_L^2 + G_R^2}{H_L + H_R + \lambda}] \tag{18}$$

where G_L represents the sum of gradients of the leaf child node, and G_R represents that of the right child node. H_L and H_R represents the sum of second-order derivatives of leaf child node and right child node, respectively. λ is the regulation constant.

This customised gain formula can be applied to calculate the score for split points. In order to find the best split over all the candidates by using Equation (18), XGBoost employs the Exact Greedy Algorithm [6]. Exact Greedy Algorithm first sorts the data regarding feature values, then enumerates all the possible splits over all the features, calculates and saves the gradient and the second-order derivative for each split, and finally finds the split with the largest gain.

To form a tree, Exact Greedy Algorithm employs a greedy strategy to determine the best split point. This algorithm is highly accurate since it enumerates all the possible split points. However, this method may introduce massive memory overhead since the data need to be sorted and stored in memory to find the best split point across a continuous feature. As a result, when dealing with a large-scale dataset or under a distributed circumstance, the efficiency of the Exact Greedy Algorithm significantly plummets. In order to solve this problem, XGBoost introduced the Approximate Algorithm [6]. The goal of the Approximate Algorithm is to quantile the dataset rather than enumerating all the split points to find the best one [27], hence reduces the computational complexity.

Specifically, the Approximate Algorithm maps all the split points that originally need to be traversed to a certain number of candidates split points according to the distribution of feature values. This process can be done by utilising the Rank Algorithm [6] that is proposed in XGBoost. A set of candidate split points can be represented as $\{x_{1,k},\ x_{2,k},\ x_{3,k},\ldots,\ x_{l,k}\}$, where k stands for the k-*th* feature, l stands for the maximum number of candidate split points. After obtaining the candidate split points, we can distribute each feature value of the feature to a bucket according to the candidate split points. Then, we sum up the gradients and second-order derivatives for each bucket, which means each bucket of the feature has cumulative gradients and second-order derivatives. Finally, the best split point is found on these cumulative statistics.

Training GBDTs with LightGBM

As we discuss above, XGBoost uses the Exact Greedy Algorithm and Approximate Algorithm to create a decision tree, which requires all the features to be pre-sorted according to the feature values. The advantage of using the pre-sorting algorithm is that it can accurately find the best split point. However, it needs to store all the feature values and pre-sorting results into the memory, which requires about twice as much memory as storing the training data. In addition, these two algorithms have to visit the entire dataset to find the best split point, which results in massive computational overhead in training time.

LightGBM [28] is an alternative implementation of GBDTs, which aims to achieve high-performance GBDT training with a large-scale dataset. LightGBM proposes several novel techniques aiming to resolve the weaknesses of XGBoost. Instead of finding the best split point by using the pre-sorting algorithm, LightGBM introduces a histogram-based algorithm [28] to obtain the best split point. The histogram-based algorithm first discretises the continuous feature values into k integers, then constructs the feature histograms with a width of k, where k also stands for the number of bins. Compared with XGBoost that uses a pre-sorting algorithm, LightGBM uses a histogram-based algorithm that only needs to calculate k bins to find the best split point. The time complexity is significantly reduced to $O(k * \#feature)$ from $O(\#data * \#feature)$, where $k << \#data$. Furthermore, a histogram-based algorithm also reduces the space complexity because it does not need to save additional information for pre-sorting results (e.g. gradients), and the size of feature values after discretisation are much smaller than continuous feature values.

XGboost and classic GBDT originally use a breadth-first growth strategy to create a decision tree. The breadth-first growth strategy splits all the leaves of the same layer, which can effectively control the complexity of the model, thereby avoiding overfitting, and has a high parallelizability. We can use multi-threads to reduce training time for this process. However, the breadth-first growth is an inefficient algorithm because it treats the leaves of the same layer indiscriminately. Some leaves may have little gains that have minimal influence on the result. Therefore, there is no need to search and split for those leaves since additional computational overhead would be added.

LightGBM abandons the breadth-first growth strategy and uses the best-first growth strategy. This technique splits the leaf with the highest gain among all the current leaves. As a result, as compared to breadth-first, best-first tend to reduce errors to achieve higher accuracy with the same number of splits. However, best-first is more likely to overfit the data because it is easy to grow a relatively high-depth decision tree. Therefore, LightGBM adds a hyperparameter called maximum depth to limit the depth of a decision tree, and thus mitigates overfitting.

3.2 PSO for Decision Trees Training

Current decision tree training methods mostly make use of the greedy algorithm as the feature selection method for the decision tree. Although it indeed accelerates the process of creating a decision tree, it makes a decision without looking ahead to see if that decision is the best in the long term. Some researchers have applied swarm intelligence to tackle this problem, which may induce a near-optimal solution. In recent years, numerous swarm-based algorithms have been introduced to construct an optimal decision tree. For example, Boryczka and Kozak proposed Ant Colony Decision Trees (ACDT)[7] in 2010, Bida and Aouat proposed Bat Tree-based Constructor (BTC) [29] in 2018. In this subsection, we focus on reviewing two novel PSO-based tree induction algorithms: Tree Swarm Optimisation

(TSO) and a pre-trained decision tree with Adaptive Particle Swarm Optimisation (APSO).

Tree Swarm Optimisation (TSO)

TSO [8] is a PSO-based algorithm, and it aims to discover the better tree structure of a symbol tree by using the PSO algorithm. Symbol tree represents any trees whose nodes contain symbols (e.g. parse trees, decision tree, operator tree). However, in the PSO algorithm, each particle's velocity and position information are expressed in the form of vectors. Therefore, we must first represent a decision tree as vectors to feed it into the PSO algorithm.

Let L denote the layer of a tree, and A denote the arity of a tree. A full tree $T_{(3,2)}$ is displayed in the Figure 2, where $T_{(3,2)}$ indicates a 2-*ary* tree with 3 layers. We can begin by calculating the size of the array using the formula as follows.

$$size(T_{(L,A)}) = \sum_{i=0}^{l-1} A_i \tag{19}$$

To convert $T_{(3,2)}$ to a vector, we can then use breadth-first traversal to visit the tree from top to bottom and calculate the node's index in the corresponding array. Let c denote the *c-th* child of a parent p, where $1 \leq c \leq A$ and $1 \leq p \leq size(T_{(L,A)}) - 1$. Given a child node, we can calculate the index in the corresponding array as follows.

$$child\_node\_index(A, p, c) = Ap + c \tag{20}$$

After that, a tree can be mapped to an array, as shown in the right of Figure 2.

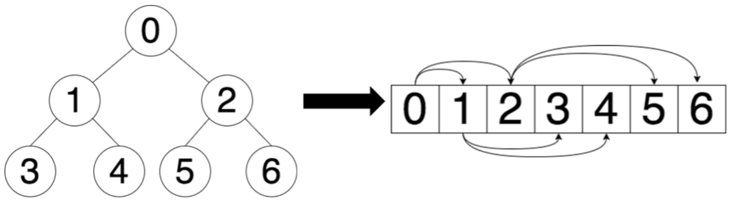


Figure 2: Mapping a full 2-ary tree with three layers to a corresponding array.

However, the main issue with the decision tree representation is determining how to convert the rules to continuous values. In order to address this issue, TSO introduces the Symbol Representation method [8]. The Symbol Representation method employs a symbol vector, the length of which is equal to the entire amount of symbols that can be represented in a node. In the case of the decision tree, this would be the summation of the total number of features and the number of classes. For instance, suppose we have a training dataset with 3 features and 3 classes, then 6 symbols are generated to represent a node, where the first 3 symbols refer to which feature to use in the rule, while the last 3 symbols refer to which class to assign.

By using the symbol vector, a decision tree $T_{(3,2)}$ with 6 symbols can be represented as the matrix below, where each column stands for a node, each row stands for a symbol (a feature or a class), and G represents the symbol scores. The symbol vector for a node j is determined by selecting the maximum symbol scores G of the column j.

$$\begin{array}{c} \\ s1 \\ s2 \\ s3 \\ s4 \\ s5 \\ s6 \end{array} \begin{array}{c} \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} G_{1,1} & G_{1,2} & G_{1,3} & G_{1,4} & G_{1,5} & G_{1,6} & G_{1,7} \\ G_{2,1} & G_{2,2} & G_{2,3} & G_{2,4} & G_{2,5} & G_{2,6} & G_{2,7} \\ G_{3,1} & G_{3,2} & G_{3,3} & G_{3,4} & G_{3,5} & G_{3,6} & G_{3,7} \\ G_{4,1} & G_{4,2} & G_{4,3} & G_{4,4} & G_{4,5} & G_{4,6} & G_{4,7} \\ G_{5,1} & G_{5,2} & G_{5,3} & G_{5,4} & G_{5,5} & G_{5,6} & G_{5,7} \\ G_{6,1} & G_{6,2} & G_{6,3} & G_{6,4} & G_{6,5} & G_{6,6} & G_{6,7} \end{pmatrix} \end{array} \tag{21}$$

Once mapping the decision tree to vectors, TSO can optimise the tree structure using the PSO algorithm. The process can be simplified into 2 steps. In Step 1, we initialise the positions and velocities of the particles randomly. In Step 2, we move the particles and evaluate their positions by the fitness function, then compare to the gbest and $pbest_i$, where $gbest$ represents the best-found tree globally, and $pbest_i$ represents the best-found tree locally. Lastly, we update the position and the velocity of the particles by Equation (15) and Equation (16).

In order to evaluate each particle, TSO utilises the number of misclassified instances as the fitness function. Specifically, TSO first decodes the vectors to a standard decision tree, then uses the decision tree to fit the multiple training data. If the decision tree predicts an incorrect classified outcome, the total number of misclassified records is increased. To evaluate the performance of the TSO algorithm for constructing a decision tree, the author has used the IRIS dataset to compare with 2 novel methods, C4.5 [17] and the Genetic Programming (GP) approach [30]. All the three methods provide close misclassification rates. TSO is 5.84, C4.5 is 5.9, and GP is 5.6. However, compared with the two swarm-based approaches, TSO has shown its advantage by costing only 2800 iterations to generate the final decision tree, while GP requires approximately 19900 iterations.

Pre-trained Decision Trees with APSO

Cho [9] have proposed a method combining greedy algorithm and APSO algorithm to construct a decision tree, which may reduce the risk of reaching a local optima solution. In order to explain this method clearly, we briefly review the APSO algorithm first.

Parameters selection has been the bottleneck affecting the global optimality and convergence speed of the PSO algorithm. To overcome this bottleneck, some researchers have integrated different adaptive methods to the PSO algorithm, and it turns out to be a successful idea to improve the performance of PSO [31], [32]. Cho [9] have applied the APSO algorithm proposed by Zhang [32] to optimise the decision tree structure. APSO introduces a control strategy for optimising ω, $c1$ and $c2$ (as discussed in Section 2.2). This control strategy is determined by the evolution factor of each particle, where evolution factor is used to estimate 4 evolution states: exploration, exploitation, convergence, and jumping out. To calculate the evolution factor, we must first compute the mean distance of the *i-th* particle to all the other particles at each iteration. The formula is displayed as follows.

$$d_i = \frac{1}{M-1} \sum_{j=1, j\neq i}^{M} \sqrt{\sum_{k=1}^{N} p(i,k) - p(j,k)^2} \tag{22}$$

where M stands for the population size, N stands for the dimensions of an optimisation problem and p represents the position. Then, let d_g denote the global best-found particle of d_i, and let d_max and d_min represent the maximum and minimum distance. Evolution factor E can be computed by:

$$E = \frac{d_g - d_{min}}{d_{max} - d_{min}} \tag{23}$$

where $E \in [0, 1]$. Once the evolution factor is gained, APSO can use the control strategy to update the parameters of each particle.

The method of creating an optimal decision tree can be summarised into 2 main steps. In Step 1, constructing a decision tree using CART, and in Step 2, optimising the pre-trained decision tree by the APSO algorithm. In this method, the authors have applied classification accuracy as the fitness function to evaluate the particles, and each particle represents a threshold of the feature rather than a single decision tree (TSO algorithm). However, when compared to the pure CART decision tree model, experiments show that the proposed method outperforms the CART.

4 Review Conclusion and Objectives

All the state-of-the-art GBDT implementations (e.g., XGBoost) in the world are currently based on the greedy algorithm to grow the decision trees, albeit with different optimisation strategies. Despite the fact that those implementations have considerably improved model accuracy and reduced training time, the greedy approach is still regarded as a short-sighted strategy for building a GBDT model. In recent years, most researchers have focused on optimising the hyper-parameters of existing GBDT approaches using swarm intelligence, with the goal of improving the prediction model's accuracy rather than the global optimality of GBDTs. Some researchers, however, have attempted to use swarm intelligence to enhance the global optimality of a single decision tree model and have obtained a satisfying result. The PSO method, in particular, has demonstrated considerable improvements in the process of searching for the optimal decision tree. As a result, in this research project, we propose to use a PSO-based approach to train the GBDT model, which may induce an optimal solution or a better solution than the greedy-based approaches.

We specify our objectives for the research project as follows.

1. Developing a robust framework to integrate the PSO algorithm to train the GBDT model.
2. Conducting experiments to compare the predictive performance with the state-of-the-art GBDT implementations.

5 Methodology

As we elaborated in Section 2.3, the GBDTs is built upon a series of decision tree. A decision tree induction can be summarised by two main actions: feature selection and node splitting. However, current state-of-the-art decision tree induction methods are mainly based on greedy algorithms and only differ in the way they define the heuristic rules (e.g., Entropy, Gini index). These heuristic rules determine which feature to apply and which nodes to cut. As we mentioned in Section 2.3, each decision tree in the GBDTs is fitted to predict the residuals rather than the actual labels. The residuals only matter with the calculations of the negative gradients of the loss function. Hence, the PSO-based search methods can be applied to search the combinations of the nodes and construct these nodes into multiple decision trees in order. Then we can traverse the decision trees to obtain the predicted results and use the results to evaluate the particle. In this section, we propose two novel PSO-based approaches for inducing the GBDTs, which differ in terms of the search space and the split quality. We call the approaches SGBDT and PSGBDT. We elaborate on the related details of the two methods in the following subsections.

5.1 GBDTs Representations

To use the PSO for searching the optimal GBDTs solutions, we need to figure out how to represent the GBDTs in a way that the PSO algorithm can understand. In Section 3.2, we review the tree representation method of the TSO algorithm. TSO algorithm uses the symbol vectors to represent a decision tree. Each node of the decision tree is represented by a symbol vector with the length of $(Num\ of\ features)$ * $(Num\ of\ labels)$. However, this representation is wasteful in terms of the memory footprint and computational complexity. Especially when training on the datasets with extremely high dimensions. Moreover, this representation method does not well fit for representing the GBDTs since GBDTs use the regression trees as the weak leaner, but the representation method is built upon only the classification problem. To address this problem, we propose to use one sequence to represent a decision tree rather than multiple symbol vectors.

Encoding

This method can be summarised in the following steps. (i) Pre-process the training dataset and extract the distinct feature values of its corresponding feature. (ii) Bind the feature, and it corresponds values into multiple tuples. (iii) Encode each tuple into a unique number or charter. (iv) Make an index table to store the tuples and their index, and use the index to represent the node.

Table 3: An example training dataset

	F1	F2	F3
0	0.1	100	-1.8
1	0.2	100	2.1
2	0.3	130	3.4
3	0.8	199	-1.7

Assume we have a training dataset that is shown in the Table 3. This dataset is made of three features and four instances. After performing the first and second steps, the unique tuples can be represented in Table 4.

Table 4: Unique Tuples

	F1	F2	F3
0	(F1, 0.1)	(F2, 100)	(F3, -1.8)
1	(F1, 0.2)	(F2, 130)	(F3, 2.1)
2	(F1, 0.3)	(F2, 199)	(F3, 3.4)
3	(F1, 0.8)	-	(F3, -1.7)

Once we obtain the multiple unique tuples that are made up of a feature and a threshold value, we can conduct the third and fourth steps to generate the index table. The corresponding index table is shown in the Table 5.

Table 5: Index table for encoding

Index	Tuples
0	(F1, 0.1)
1	(F1, 0.2
2	(F1, 0.3)
3	(F1, 0.8)
4	(F2, 100)
5	(F2, 130)
6	(F2, 199)
7	(F3, -1.8)
8	(F3, 2.1)
9	(F3, 3.4)
10	(F3, 3.4)

Tree Representation by Index Table

Suppose we define a full decision tree with depth 4, and with a set of decision rules (F2,130), (F1,0.8), (F1,0.2), (F1,0.1), (F3,0.3), (F2,100), (F3, -1.8). The decision tree with the internal nodes can be visualised in Figure 3.

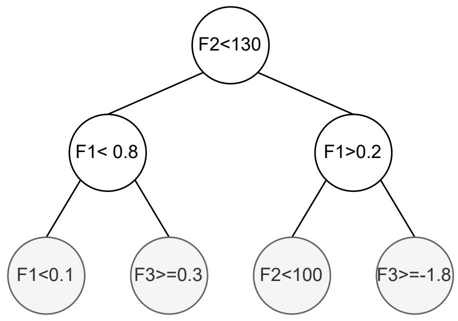


Figure 3: An unconstrained Decision Tree

We first breadth-first traverse the decision tree, and then look up the index of the node from the index table (Table 5). For example, the root node with the rule (F2, 130) is mapped to 5 from the index table. So, we use 5 to represent the current node. After traversing the tree, the sequence used to represent this decision tree is [5, 3, 1, 0, 2, 4, 7].

For representing the GBDTs, we can use the index table to encode every decision tree into multiple sequences. And then concatenate the multiple sequences into one single sequence.

5.2 Oblivious Decision Tree as Weak Learners

In section 3.1, we discuss the two popular implementations of the GBDTs, XGBoost and LightGBM. XGBoost optimises the traditional GBDTs in terms of accuracy, speed, and generalisation ability. LightGBM optimises the XGBoost to obtain higher speed and better memory footprint. These two state-of-the-art implementations apply the unconstrained decision trees as the weak learners. Even though using unconstrained decision trees can indeed improve the learning performance on the training set, this tree structure suffers the risk of overfitting. Prokhorenkova et al. [33] have proposed a new gradient boosting method based on the Oblivious Decision Trees (ODTs) named CatBoost. Researchers have demonstrated the good performance of this method in terms of the fast inference and generalisation ability [33], [34]. We explain the rationale of the oblivious tree and how does it use to achieve faster inference in the following text.

ODT

For an unconstrained DT, each node splits into 2 branches: internal node or leave. Unlike the unconstrained DT, ODT has 2 main characteristics that are different to an unconstrained DT. (i) ODT is a full decision tree, each node of the ODT splits into 2 branches that are either internal nodes or leaves. (ii) Each layer of ODT has the same decision rules. Obviously, ODTs are significantly weak learners since we constrain the tree structure for the splits. However, when applying on the Gradient Boosting learning, such weakness can be seen as adding regulation terms to avoid the risk of overfitting. Figure 4 shows a simple ODT with 3 layers.

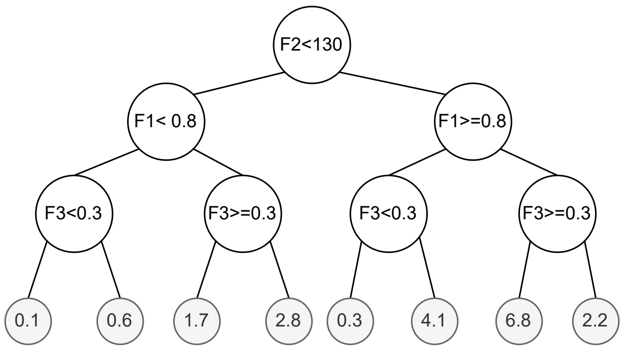


Figure 4: An ODT with 3 layers

Inference

We can use a table with 2^d entries to represent an ODT, where d represents the depth of the tree, 2^d represents all possible combinations of d splits. Figure 4 shows 8 (2^{d-1}) leaves at the bottom of the tree. We can index these 8 nodes from 0 to 7, and each leaf has a corresponding value. Hence, when predicting an instance for a tree, we only need to find the corresponding index of the instance. Due to the decision rules of each layer in the oblivious tree is the same, each layer can be represented by a set of binary code (0 or 1). Take the second layer from Figure 4 for example, if the F1 $<$ 0.8, encode this layer as 1, else 0.

Example

In this subsection, we make an intuitive example to present the inference process of the ODT. First, we can make an index table for the 8 leaf values at last layer.

Table 6: Index table of leaf values for Figure 4

Index	0	1	2	3	4	5	6	7
Value	0.1	0.6	1.7	2.8	0.3	4.1	6.8	2.2

For the instances from the example training dataset (Table 3), we can encode each instance to a binary code. For example, suppose we have a data instance {(F1, 0.1), (F2, 100), (F3, -1.8)}. We get the encoded value as "101", because

$$1^{st}\ layer : (F2 = 100) < 130 \rightarrow True \rightarrow 1 \rightarrow right$$

$$2^{nd}\ layer : (F1 = 0.1) >= 0.8 \rightarrow False \rightarrow 0 \rightarrow left$$

$$3^{rd}\ layer : (F3 = -1.8) < 0.3 \rightarrow True \rightarrow 1 \rightarrow right$$

"101" represents 5 in decimal. Hence, we can look up the value of index 5 from Table 5, and then we get 4.1. The full encoded training dataset is represented in Table 7.

Table 7: Inference for 4 training instances

	F1	F2	F3	Code	Index	Value
0	0.1	100	-1.8	101	5	4.1
1	0.2	100	2.1	100	4	0.3
2	0.3	130	3.4	011	3	2.8
3	0.8	199	-1.7	001	1	0.6

By applying the ODT as the weak leaner, the depth-first search can be replaced by array manipulation, making the inference time complexity $O(depth)$, where *depth* represents the depth of the ODT. Empirical evidence [33] demonstrates that using ODT as the weak learners can achieve tens of times faster than using the unconstrained decision tree. Therefore, we decided to use ODTs as the weak learners for our swarm-based GBDTs method because we use the accuracy and RMSE(Root Mean Square) as the fitness function to evaluate the particles, implying that each particle in the swarm needs to compute accuracy or RMSE for the prediction model at each iteration. ODT can significantly reduce the fitness calculations when setting a large population size.

5.3 Particle Modeling with Discrete PSO

In section 2.2, we review the conventional PSO algorithm. As we know, the conventional PSO algorithm is only applicable for the optimisation problem that is based on the continuous search space. However, a larger number of real-world optimisation problems are based on the discrete search space, such as TSP [35] and

0/1 Knapsack problem [36]. TSP and Knapsack problem is a well-known Combinatorial Optimization Problem (COP) which takes non-polynomial time to induce the optimal solution. And we know optimal decision tree induction is also a COP problem [4]. However, as we reviewed in section 3.2, Cho trains the decision tree with APSO using the pre-trained model [9]. Thus, the traditional PSO algorithm can be used to search for the optimal thresholds. In this way, the traditional PSO algorithm is successfully applied to the COP problem.

Nevertheless, when this method is applied to train GBDTs, it comes up with two bottlenecks. First, the tree's structure is constrained by the pre-trained model, so we cannot search for the optimal permutations. Second, the tree structure is changed if the threshold values are modified by PSO, which requires re-calculating the negative gradients for the entire dataset. Re-calculation introduces extra computational complexity, especially for big population sizes. Hence, the conventional PSO algorithm might not be an efficient method to optimise these COP problems.

Numerous Discrete PSO methods have been proposed in recent years to address the COP problems. Kennedy and Eberhart first introduced a discrete version of PSO in 1997 called BPSO [37]. Wang et al. proposed a particular discrete PSO method in 2003, which introduces the concept of swap operator and swap sequence to address the TSP problem [38]. Veeramachaneni later put forward VPSO [39], which uses the probabilistic transition rules to improve the BPSO. Inspired by the original idea of the swap operator, we propose a modified swap strategy to integrate GBDTs training with PSO. In this subsection, we explain how the particles are modelled and updated to train GBDTs.

Particle Modeling

In PSO, each particle represents a candidate solution in the search space. To find the optimal solutions for GBDTs, we can model each particle to represent GBDTs with random splits. Hence, each particle in the swarm represents a GBDT solution, and each GBDT solution is encoded by a discrete sequence we present in section 5.1. Figure 5 show an example swarm-based GBDTs model with population size n, each particle in the swarm represents a random GBDT solution with m Oblivious

Decision Trees.

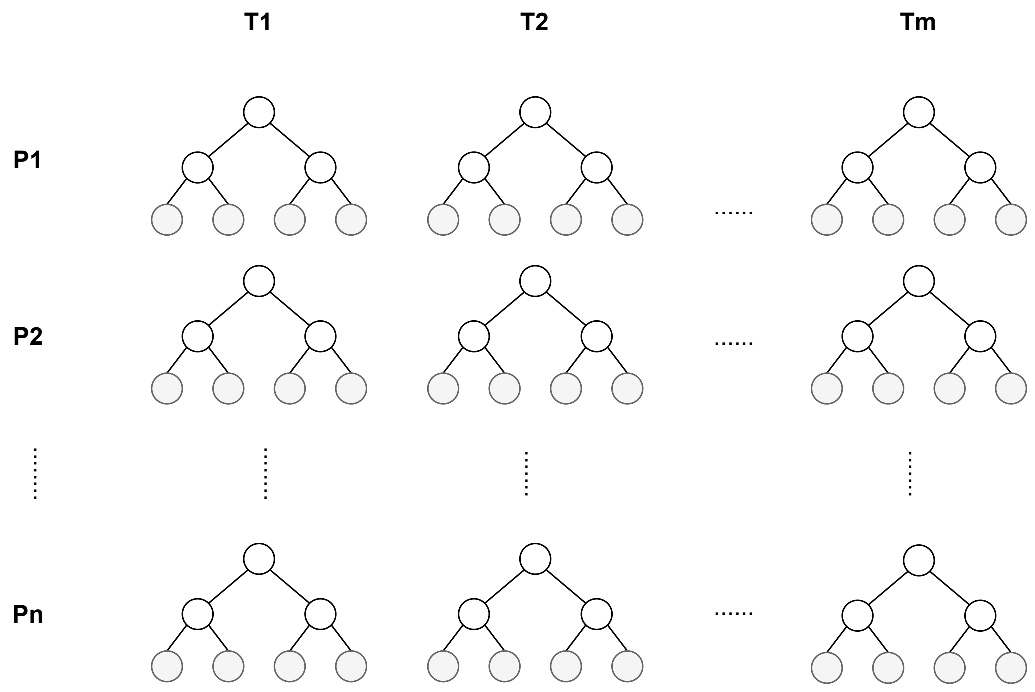


Figure 5: An example of particle modeling

Particle Update

The velocity update formula (Equation 15) of the PSO requires to calculate $pbest_{i,d} - p_{i,d}$ and $gbest_d - p_{i,d}$. However, the particle position is a GBDT solution encoded by a discrete sequence, where the encoded elements contain the feature information of the internal nodes. Acting a direct subtraction on two discrete sequences becomes meaningless because there is no sensible relationship between elements from the sequence, even though we can encode the nodes to integers. To address the issue, we modify the velocity equation as follows.

$$v_{(i,d)} = v_{(i,d)} \oplus \alpha(pbest_{i,d} - p_{i,d}) \oplus \beta(gbest_d - p_{i,d}) \tag{24}$$

Where α and β are random weights between 0 and 1, they control the randomness of the particle explorations. Also, α and β determine which position particles want to approach closer (global best position or local best position?). For calculating $pbest_{i,d} - p_{i,d}$ and $gbest_d - p_{i,d}$, we use velocity operator to describe the subtraction of two discrete sequence. If the position values in each dimension are not the same, we add position information to the velocity operator.

Specifically, assume we have the current particle's position p_i = [2, 5, 19, 8, 7, 1], local best position $pbest_i$ = [3, 56, 18, 8, 0, 3], and global best position $gbest$ = [4, 8, 10, 87, 4, 5]. We set $OP1 = pbest_i - p_i$, $OP2 = gbest - p_i$. Then we calculates $OP1$ and $OP2$ as follows.

$$\begin{aligned} OP1 \rightarrow \{&V(0, pbest_i(0), \alpha), V(1, pbest_i(1), \alpha), V(2, pbest_i(2), \alpha), \\ &V(4, pbest_i(4), \alpha), V(5, pbest_i(5), \alpha)\} \end{aligned}$$

$$\begin{aligned} OP2 \rightarrow \{&V(0, gbest(0), \beta), V(1, gbest(1), \beta), V(2, gbest(2), \beta), \\ &V(3, gbest(3), \beta), V(4, gbest(4), \beta), V(5, gbest(5), \beta)\} \end{aligned}$$

We merge $OP1$ and $OP2$ by $\oplus$.

$$\begin{aligned} OP1 \oplus OP2 \rightarrow \{&V(0, pbest_i(0), \alpha), V(1, pbest_i(1), \alpha), V(2, pbest_i(2), \alpha), \\ &V(4, pbest_i(4), \alpha), V(5, pbest_i(5), \alpha), V(0, gbest(0), \beta), \\ &V(1, gbest(1), \beta), V(2, gbest(2), \beta), V(3, gbest(3), \beta), \\ &V(4, gbest(4), \beta), V(5, gbest(5), \beta)\} \end{aligned}$$

$OP1 \oplus OP2$ is the updated velocity of the current particle. It is described by a set of velocity operators, each velocity operator contains the *d-th* dimension, correspond position (*pbest* or *gbest*), and its random weight. To calculate the new position of the current particle, we iterative process the velocity operators, and then swap *d-th* dimension of the p_i to correspond $pbest_i$ or $gbest$ respect to the random weight. Take the first velocity operator from $OP1 \oplus OP2$ as example, we first generate a random number between [0, 1], denote as $random$. if $random < \alpha$, we swap $pbest_i(0)$ to $p_i(0)$. Now the new particle position p'_i becomes [3, 5, 19, 8, 7, 1]. We elaborate the details of this particle update algorithm in Algorithm 2.

Algorithm 2: Algorithm for particle update

```
Input  : Number of iterations $Iters$
         List of particles $Pars$
         Random weight for local best $\alpha$
         Random weight for global best $\beta$
1  The method **GetGbest** returns the global best particle position, **GetPbest** returns
     the local best particle position, and **GetCurrent** returns the current particle
     position. **VOP** is a list to save the velocity operators
2  Function Update
3      for $iter \in Iters$ do
4          $Gbest \leftarrow$ GetGbest($Pars$) ;
5          for $par \in Pars$ do
6              $Pbest \leftarrow$ GetPbest() ;
7              $P \leftarrow$ GetCurrent() ;
               // Calculate $Pbest_{i,d} - P_{i,d}$
8              for $i \in P$ do
9                  if $P(i) \neq Pbest(i)$ then
10                     $V \leftarrow (i, Pbest(i), \alpha)$;
11                     Add $V$ to $VOP$;
12                 end
13             end
               // Calculate $Gbest_i - P_{i,d}$
14             for $i \in P$ do
15                 if $P(i) \neq Gbest(i)$ then
16                     $V \leftarrow (i, Gbest(i), \beta)$;
17                     Add $V$ to $VOP$;
18                 end
19             end
20         end
21     end
22     for $V \in VOP$ do
23         if $random \leq V(2)$ then
24             Swap($P(V(0))$, $V(1)$);
25         end
26     end
```

5.4 Training GBDT with Discrete PSO

In this subsection, we explain the SGBDT and PSGBDT algorithm. Training GBDTs using PSO can be explained in 3 major steps : swarm initialisation, swarm evaluation, and swarm update. In the following text we will elaborate the details for swarm initialisation and evaluation. The swarm update algorithms is discussed in Section 5.3. Then, we describe the entire training pipeline of the SGBDT and PSGBDT algorithm .

Initialisation

As we discuss in section 5.3, each particle in the swarm represents a random GBDTs solution. Hence, particle initialisation can be seen as the process of generating a bunch of GBDTs solutions with random feature selections. Because we use the ODTs as weak learners, we only need to generate the random rules for each layer of the decision tree. For a GBDTs with m estimators and d depth, we generate $(m * d)$ random rules for initialising a particle. The complete initialisation process is described in algorithm 3 and 4. We skip the explanation of constructing GBDTs because we explain the relevant details in section 2.3. However, we do specify the two differences between the SGBDT training and the standard GBDT training. First, we use ODT as the weak learners rather than the CART regression trees. Second, we do not use greedy algorithms to find the best split point at each internal node. Instead, we grow the trees with a set of arbitrarily given rules.

Algorithm 3: Algorithm for Particle Initialisation

```
Input  : Population size $L$
The method GetRandomGBDTs is described in Algorithm 4
Function Init
    for $l \in L$ do
        $seq$ ← GetRandomGBDTs();
        BuildGBDTs($seq$);
    end
```

Algorithm 4: Algorithm for Random Features Selection

```
Input  : Dataset $S$
         Number of trees $m$
         Tree depth $d$
Output: Encoded value $\sigma$
1 The method Rand returns a random feature index, RandVal returns a
    random threshold for the correspond feature, GetIndex returns the encode
    value of a random rule from the index table (Section 5.1)
2 Function GetRandomGBDTs
3     for $i \leftarrow 0$ to $m * d$ do
4         $f \leftarrow$ Rand($S$);
5         $val \leftarrow$ Rand($S$, $f$);
6         $\sigma \leftarrow$ GetIndex($f$, $val$);
7         Add $\sigma$ to $O$ ;            // $O$ is a list to save the random rule
8     end
9     return $O$;
```

Evaluation

Given a state of a swarm, we need to evaluate the quality of each particle so that we can update the local best solution and global best solution. In this case, particle quality can be determined by the performance of the GBDTs solution. To evaluate the GBDTs performance, we use accuracy and RMSE as the metric. Hence, particle evaluation is seen as the process of GBDTs inference. For standard GBDTs, inference usually involves two actions: decision tree inference and model summation. We discuss the relevant details for ODTs inference in Section 5.2. And the model summation is conducted by Equation 6.

SGBDT Training Pipeline

The entire training process of SGBDT is illustrated in Figure 6. We describe the entire training process as the following steps. (i) We create a swarm with population size L, Iterations I, random weight α and β, number of trees N, and tree depth d. (ii) Initialise the swarm using Algorithm 3. (iii) Encode the swarm by index table. (See deatails in Section 5.1).(iv) Evaluating the swarm by calculating the training accuracy or RMSE of the particles. (vi) Updating the *Gbest* solution after evaluation if the current solution has a better fitness value (or accuracy) than the last iteration. (vii) Updating the swarm using Algorithm 2 until the training reaches the maximum iterations I.

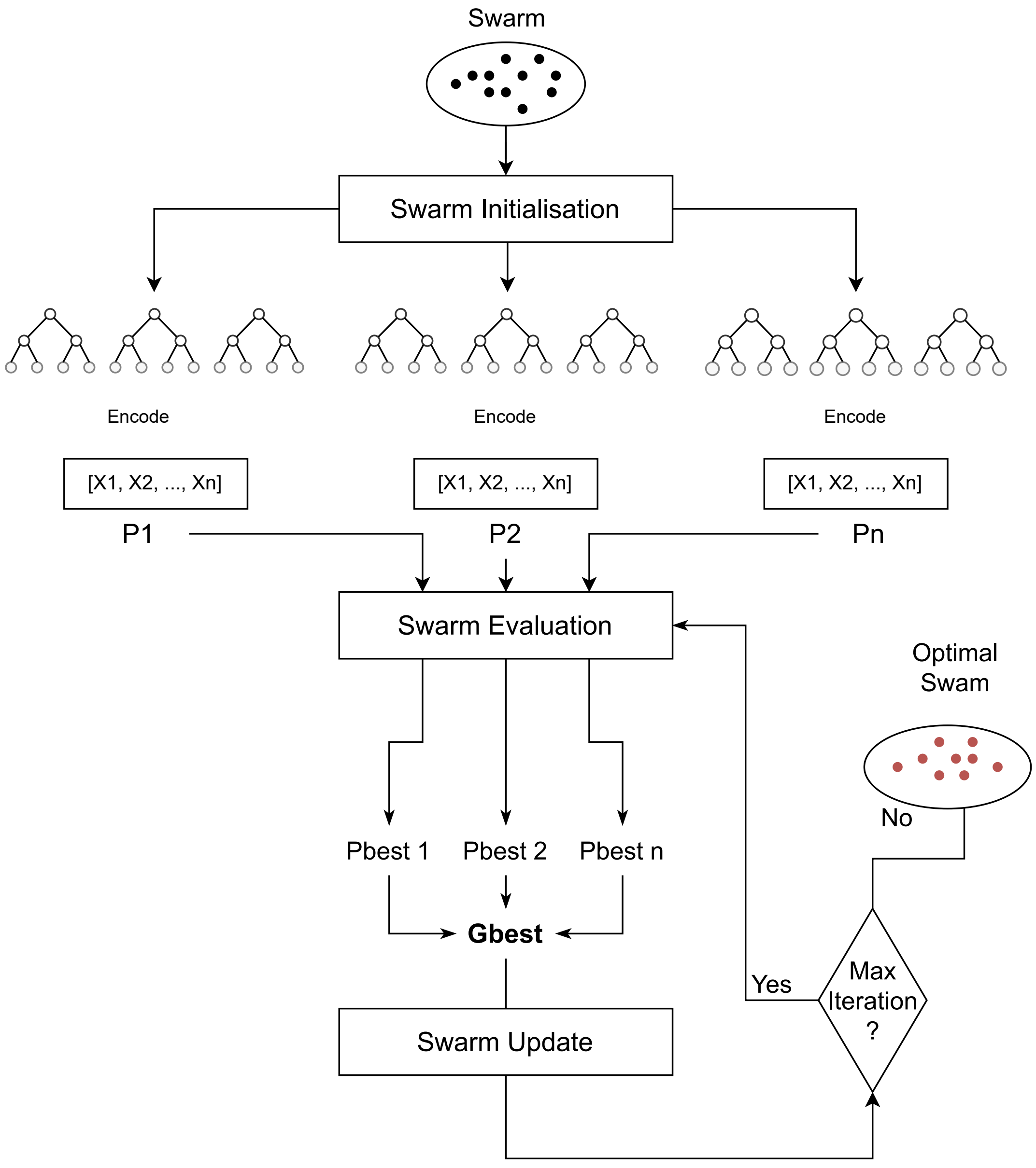


Figure 6: SGBDT training pipeline

Discretisation

LightGBM [28] applies feature discretisation to reduce the time and space complexity of the GBDTs training. Learning from the idea, we integrate feature discretisation into SGBDT training to reduce the search space rather than computational complexity.

The search space of the particle can be computed by the index table, which is explained in section 5.1. The index table includes every distinct feature and threshold of the training dataset. Assume we have an index table with n instances. And we set the tree's number as m, and the tree's depth as d. We can define the problem as r-permutations of n [40].

Given a set of elements with length n, we randomly pick up r-element subset of an n-set in different ordered arrangements, where $r < n$. We can denote the number of r-permutations of n by P_n^r, and its value is given by the following formula [40].

$$P_n^r = n * (n-1) * (n-1) ... (n-r+1) = \frac{n!}{(n-r)!} \tag{25}$$

In this case, we randomly pick up r-elements from the index table, where r represents how many decision rules a GBDT model has. In an ODT, each internal layer has only one decision rule. Hence, $r = m * (d-1)$, where $(d-1)$ stands for the number of internal layers of an ODT. Therefore, We can represent the search space S of the particle as follows.

$$S = P_n^r = \frac{n!}{[n - m * (d-1)]!} \tag{26}$$

Feature discretisation can significantly reduce the size of n by discretising continuous values of features to k bins, where $k << n$. For example, suppose we have a dataset N with 500 instances and two numeric features. And we set $m = 3$, $d = 4$. Assume all thresholds of two features are distinct, $n = 500 * 2 = 1000$. We have search space:

$$S_1 = \frac{1000!}{[1000 - 3 * (4-1)]!} = \frac{1000!}{991!} \approx 9.64 * 10^{26}$$

Assume we discretise 500 continuous values to 8 bins, $n = 8 * 2 = 16$. We have search space:

$$S_2 = \frac{16!}{[16 - 3 * (4 - 1)]!} = \frac{16!}{7!} \approx 4.15 * 10^9$$

The search space is massively decreased from $9.64 * 10^{26}$ to $4.15 * 10^9$ using feature discretisation.

Training with PSGBDT

SGBDT algorithm randomly selects the decision rules from the search space and uses the specified rules to construct the GBDT model for each particle. Obviously, the search space is too massive for PSO to converge on a reasonable solution, even though we can shrink the search space through discretisation. Furthermore, random initialisation may result in the construction of numerous trees with extremely high loss. Therefore, we propose using the split points pre-trained by greedy GBDT methods to train our SGBDT algorithm, reduce the search space further, and improve the quality of the split points. We name the technique PSGBDT.

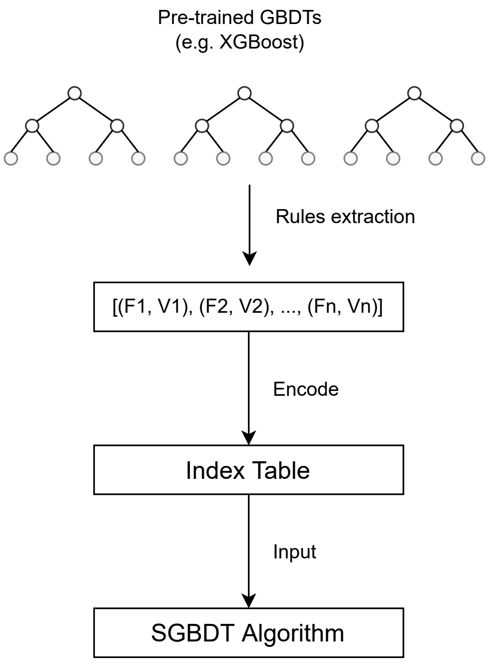


Figure 7: PSGBDT training pipeline

The PSGBDT algorithm is built based on the SGBDT algorithm (Algorithm 6). In the SGBDT algorithm, we initialise particles with arbitrary decision rules selected from the index table. However, we initialise particles with the pre-trained split points in the PSGBDT algorithm. We describe the process of the PSGBDT algorithm in the following steps. (i) Training a GBDTs model using state-of-the-art libraries such as XGBoost, LightGBM and CatBoost. (ii) Extracting the decision rules from the tree's nodes. (iii) Encoding the decision rules to an index table. (iv) Input the index table to the SGBDT algorithm to initialise particles with the pre-train split points. We illustrate the training pipeline of the PSGBDT algorithm in Figure 7

6 Experimental Objectives Description

From section 6, we propose two Swarm-based algorithms to induce the GBDTs: SGBDT and PSGBDT. In this section, we identify multiple objectives for the experiments.

1. How does SGBDT and PSGBDT compare to the state-of-the-art tree-based learning approaches in terms of predictive performance and global optimality?

2. We provide the option to use the discretisation algorithms to reduce the search space of the particle. So, how does SGBDT perform if we apply the discretisation algorithms? Does it improve the predictive performance?

3. How does PSGBDT compare to SGBDT in terms of the predictive performance since PSGBDT can further reduce the search space and search with the split points pre-determined by the greedy GBDT algorithms?

7 Results

In this section, we seek to answer the objectives that are elaborated in section 6 by experiments. We present the experimental results of the three proposed methods testing on various classification and regression dataset. The three methods are SGBDT, SGBDT-bins and PSGBDT. SGBDT-bins is to use SGBDT with discretisation, and PSGBDT integrate the pre-trained results from XGBoost. The description of the datasets is presented in Table 8.Then, we compare the experimental results generated by our three swarm-based methods to other 4 mainstream libraries; that is CART Decision Tree, GBDT (Sklearn), XGBoost and CatBoost. In addition, we present the view of particle explorations during the SGBDT and PSGBDT training.

We use the datasets that are used in [6], [41], [42], and the rest datasets are retrieved from UCI Machine Learning Repository [43] and Kaggle [44], [45]

Table 8: Datasets description

Dataset	Number of Instances	Number of Features	Type	Metric
BankNotes [43]	1372	4	Classification	Accuracy
Wine [41]	1599	11	Classification	Accuracy
Covertype [42]	58101	55	Classification	Accuracy
Higgs [6]	55000	29	Classification	Accuracy
Kc_house_data [44]	24768	19	Regression	RMSE
Insurance [45]	1338	7	Regression	RMSE
Red_wine_quality [41]	1385	12	Regression	RMSE

7.1 Compared with Benchmarks

For the environment setting, we set the tree depth as 7 for CART training. The rest of methods are trained with estimators = 6, depth = 5, learning_rate = 1. For the three swarm-based methods, we set α and $\beta = 0.45$. Population size is set to 50 for classification tasks and 200 for regression tasks. We set max_bin size as 100 for SGBDT-bins, implying all features will be discretised to 100 bins.

Table 9 illustrate the classification accuracy and RMSE over the seven sample datasets. Table 10 describes the average classification accuracy and RMSE over the seven sample datesets. The accuracy scores and RMSE generated by all methods are obtained by the average values of ten times running on the test sets.

Table 9: Average test accuracy and RMSE of benchmarks using ten measurements

Dataset	CART	Sklearn	XGBoost	CatBoost
BankNotes	0.98 ± 0.01	**0.99 ± 0.01**	**0.99 ± 0.01**	0.98 ± 0.02
Wine	0.70 ± 0.05	0.74 ± 0.04	0.74 ± 0.06	0.74 ± 0.04
Covertype	0.88 ± 0.01	**0.90 ± 0.01**	**0.90 ± 0.01**	0.87 ± 0.01
Higgs	0.68 ± 0.01	**0.69 ± 0.01**	**0.69 ± 0.01**	0.68 ± 0.01
Red_wine	0.57 ± 0.05	**0.56 ± 0.04**	0.56 ± 0.05	0.63 ± 0.05
Kc_house	1.89E+05 ± 2.62E+04	1.83E+05 ± 2.59E+04	1.77E+05 ± 1.74E+04	1.85E+05 ± 1.58E+04
Insurance	4.24E+03 ± 5.64E+02	4.18E+03 ± 6.45E+02	4.33E+03 ± 5.83E+02	4.74E+03 ± 6.74E+02

Table 10: Average test accuracy and RMSE of swarm-based methods using ten measurements

Dataset	SGBDT	SGBDT-bins	PSGBDT
BankNotes	0.98 ± 0.02	**0.99 ± 0.01**	**0.99 ± 0.01**
Wine	0.74 ± 0.03	0.74 ± 0.03	**0.75 ± 0.01**
Covertype	0.86 ± 0.02	0.87 ± 0.02	0.88 ± 0.02
Higgs	0.65 ± 0.02	0.66 ± 0.01	**0.69 ± 0.01**
Red_wine	0.65 ± 0.04	0.68 ± 0.01	0.61 ± 0.01
Kc_house	2.04E+05 ± 9.52E+03	1.82E+05 ± 2.83E+04	**1.76E+05 ± 1.83E+04**
Insurance	**4.13E+03 ± 2.57E+02**	4.85E+03 ± 8.46E+02	4.21E+03 ± 9.96E+02

All comparisons are conducted under the same hyper-parameters setting of tree depth, estimators, and learning rate. Both CatBoost and our swarm-based GBDT use ODT as weak learners. ODT trades the predictive performance in exchange for the inference speed up by constraining the decision tree structure. Hence CatBoost is the main baseline to compare with our swarm-based GBDT methods. We highlight the output data in Table 9 and 10 if it perform greatest among the seven methods.

Experiments demonstrate the traditional GBDT (Sklearn) and XGBoost remain overall the best on relatively large-scale datasets(Higgs and Covertype) in terms of the predictive performance. SGBDT and SGBDT-bins output near results with CART and CatBooost on the seven datasets. However, experiments show PSGBDT outperform traditional GBDT and XGBoost on Wine, Kc_house and Insurance dataset, and it outperforms CatBoost across all sample datasets.

Overall, SGBDT and SGBDT-bins can generate decent results on relatively small-scale datasets such as BankNotes and Wine. The predictive performance plummets significantly once the scale of the dataset increase because the search space massively increases as well. However, PSGBDT shows outstanding performance when dealing with small and large scale datasets using pre-train split points to optimise the search space. We will analyse the search space differences between the three swarm-based GBDT methods in the next subsection.

7.2 Compared Swarm-based GBDT approaches

Overall, experiments show PSGBDT has the most significant predictive performance on the six datasets, and SGBDT methods with discretisation almost always outperform SGBDT without discretisation. These experimental results provide good evidence that reducing the size of the search space can indeed improve the predictive performance of our swarm-based models. In the following text, we will analyse the differences in the search space size of the three methods based on empirical evidences.

Search Space Analysis

We explain how to represent the search space of particle using Equation 26.

$$S = \frac{n!}{[n - m * (d - 1)]!}$$

In our experiments, we set the same estimators m and tree depth d. So the major difference between three methods is the size of n. SGBDT calculate the search space S using the number of distinct values (or the length of index table). SGBDT-bins use the number of distinct bins to compute S. However, PSGBDT initialise the swarm with the pre-trained split points that are generated by XGBoost. Thus, n is reduced to the number of internal nodes in the pre-trained model. The number of nodes can be quickly addressed by extracting the XGBoost tree repositories. We measure different size of n respect to three approaches and present the results in Table 11

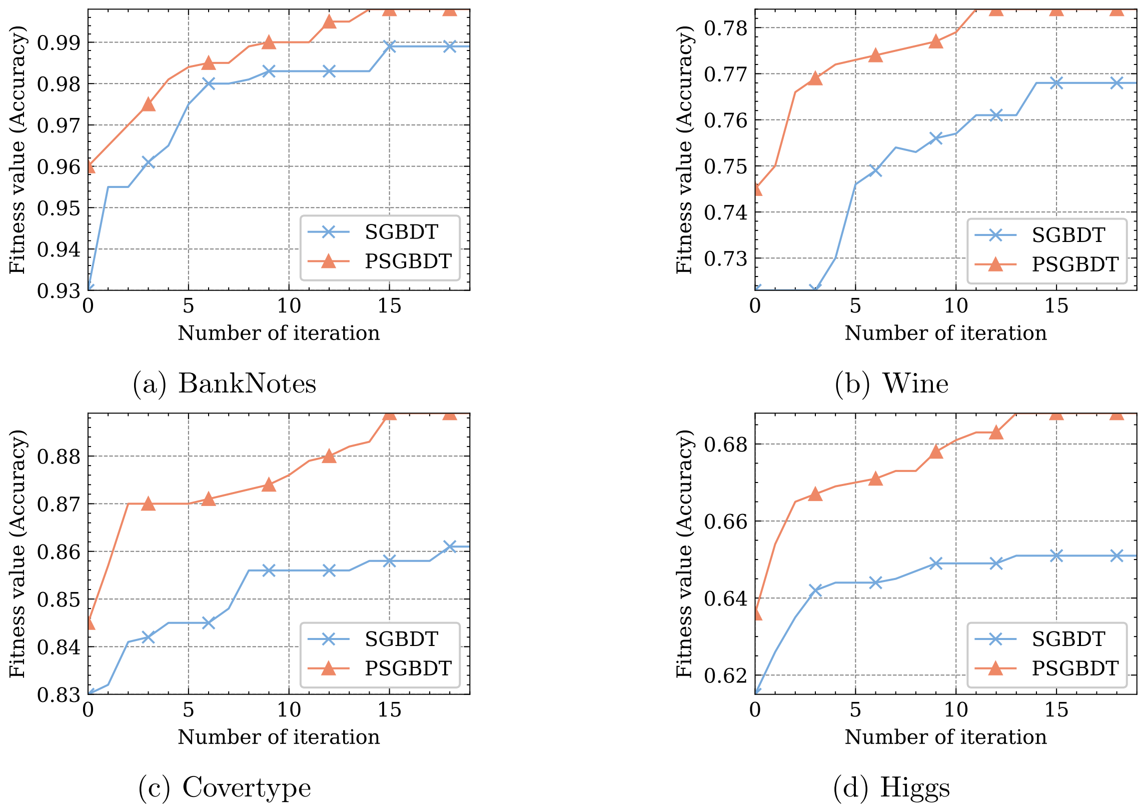
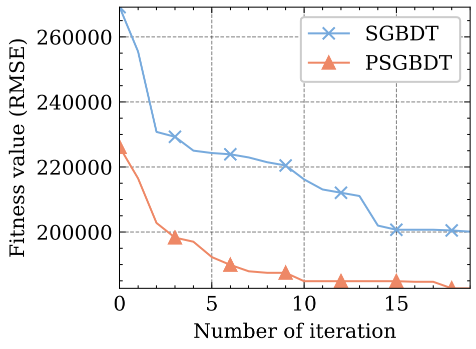


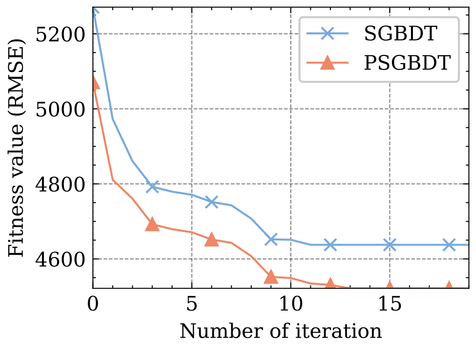
Figure 8: Particle explorations for classification tasks. Theses figures record the fly of the particles during the training process of SGBDT and PSGBDT. X-axis stands for the iteration, y-axis represents the global best accuracy at each iteration

Table 11: The size of n respect to three swarm-based approaches

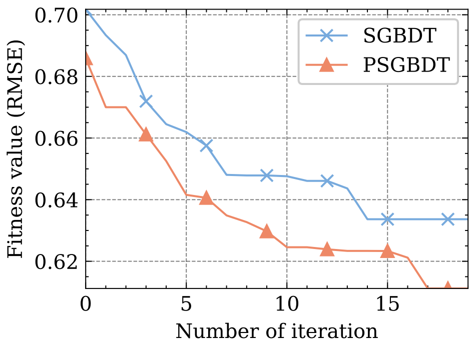
Datasets	Distinct Values	Distinct Bins	Internal Nodes
BankNotes	4082	392	61
Wine	1384	831	128
Covat	15044	896	158
Higgs	469488	2356	181
Kc_house	24768	1054	73
Insurance	555	148	145
Red_wine	1385	828	141



(a) Kc_house_data



(b) Insurance



(c) Red_wine_quality

Figure 9: Particle explorations for regression tasks. Similar to Figure 8
, but the y-axis represents the global best RMSE at each iteration

Once we obtain the size of n, we can compute the search space of three methods on seven datasets. We present the search space size in Table 12.

Table 12: Swarm-based GBDT Search Space Size

Datasets	SGBDT	SGBDT-bins	PSGBDT
BankNotes	4.2E+86	8.4E+61	3.7E+40
Wine	2.0E+75	8.4E+69	3.7E+49
Covat	1.8E+100	5.3E+70	9.3E+51
Higgs	1.3E+136	7.6E+80	3.1E+53
Kc_house	2.8E+105	2.7E+72	7.3E+42
Insurance	4.4E+65	1.7E+51	9.9E+50
Red_wine	2.0E+75	7.7E+69	4.8E+50

8 Discussion

In this section, we discuss the limitations and future works of this research project.

8.1 Limitations

Computational bottlenecks limit the particle explorations. When particles explore a new solution, they must reconstruct a GBDT to assess the quality of the present solution. Although swarm-based approaches do not require traversing the entire dataset to find the best split point, the entire dataset must be updated each time when a negative gradient is determined. As a result, the negative gradient calculations constrain the scope of particle exploration; thus, particles are more likely to be imprisoned in the local optima solution.

Swap-based PSO method may lead to early convergence. As shown in Figure 8 and 9, we can observe that particles converge too early on Insurance and Higgs during the explorations, even though we set a small number of iterations. A direct solution is to lower α and β to improve the randomness of the search. However, PSO may fail to converge at a good outcome if α and β are set too low.

8.2 Future Works

In terms of future research, we intend to look at the performance of SGBDT and PSGBDT by employing the unconstrained decision tree as the weak learners. Empirical results show that PSGBDT can outperform CatBoost on seven datasets. We believe that our swarm-based approaches can be further improved using an unconstrained decision tree because the oblivious decision tree sacrifices a small part of the predictive performance to gain faster inference speed. In addition, we plan to solve the computational bottleneck by inducing a single decision tree. Our current swarm-based GBDT methods face the bottleneck of negative gradient calculations. We plan to train a single decision tree using our swap-based PSO algorithm with the pre-trained split points. We believe the idea can solve the computational bottleneck and maintain predictive performance. Besides, the single decision tree has the advantage of interpretability, which is one field the GBDT model fails in.

Moreover, to solve the early convergence of PSO, adaptive strategy is one of the better solutions than hyper-parameters tuning. We can incorporate this method to adaptively modify α and β in each iteration by evaluating particle exploration and exploitation. Last but not least, deep Reinforcement Learning (RL) has been a popular technique to optimise the COP problems. Liang [46] demonstrates that using deep RL to induce a decision tree can significantly improve the classification time and memory footprint compared to state-of-the-art algorithms. Hence, we may want to try deep RL to induce the GBDT in future work.

9 Conclusions

In this dissertation, we present two novel approaches to training GBDT using the swarm-based search algorithm rather than the traditional greedy algorithm. Traditional greedy GBDT methods enumerates the entire dataset to find the best split points at each node when constructing a GBDT. However, swarm-based methods can skip making a greedy decision and construct a GBDT with given split points generated by particle explorations. To the best of our knowledge, this is the first attempt to employ the PSO algorithm for GBDT training, and practical results show that these techniques have good predictive performance.

Although our proposed techniques are still constrained by a computational bottleneck, we outline the associated efforts that are possible to solve the challenges and demonstrate the possibility to enhance the existing methods. We hope our proposed methods can inspire researchers to develop a new generation of non-greedy GBDT training algorithms.

Bibliography

[1] L. Li, Y. Yu, S. Bai, J. Cheng and X. Chen, 'Towards effective network intrusion detection: A hybrid model integrating gini index and gbdt with pso,' *Journal of Sensors*, vol. 2018, 2018.

[2] J. Cheng, G. Li and X. Chen, 'Research on travel time prediction model of freeway based on gradient boosting decision tree,' *IEEE access*, vol. 7, pp. 7466–7480, 2018.

[3] M. Grbovic and H. Cheng, 'Real-time personalization using embeddings for search ranking at airbnb,' in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 311–320.

[4] H. Laurent and R. L. Rivest, 'Constructing optimal binary decision trees is np-complete,' *Information processing letters*, vol. 5, no. 1, pp. 15–17, 1976.

[5] L. Breiman, J. H. Friedman, R. A. Olshen and C. J. Stone, *Classification and regression trees*. Routledge, 2017.

[6] T. Chen and C. Guestrin, 'Xgboost: A scalable tree boosting system,' in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[7] U. Boryczka and J. Kozak, 'Ant colony decision trees–a new method for constructing decision trees based on ant colony optimization,' in *International Conference on Computational Collective Intelligence*, Springer, 2010, pp. 373–382.

[8] C. Veenhuis, M. Koppen, J. Kruger and B. Nickolay, 'Tree swarm optimization: An approach to pso-based tree discovery,' in *2005 IEEE Congress on Evolutionary Computation*, IEEE, vol. 2, 2005, pp. 1238–1245.

[9] Y.-J. Cho, H.-S. Lee and C.-H. Jun, 'Optimization of decision tree for classification using a particle swarm,' *Industrial Engineering and Management Systems*, vol. 10, no. 4, pp. 272–278, 2011.

[10] R. C. Barros, M. P. Basgalupp, A. C. de Carvalho and A. A. Freitas, 'A hyper-heuristic evolutionary algorithm for automatically designing decision-tree algorithms,' in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, 2012, pp. 1237–1244.

[11] D. Jankowski and K. Jackowski, ‘Evolutionary algorithm for decision tree induction,’ in *IFIP International Conference on Computer Information Systems and Industrial Management*, Springer, 2015, pp. 23–32.

[12] D. Karaboga and B. Basturk, ‘On the performance of artificial bee colony (abc) algorithm,’ *Applied soft computing*, vol. 8, no. 1, pp. 687–697, 2008.

[13] X.-S. Yang and A. H. Gandomi, ‘Bat algorithm: A novel approach for global engineering optimization,’ *Engineering computations*, 2012.

[14] R. Eberhart and J. Kennedy, ‘A new optimizer using particle swarm theory,’ in *MHS’95. Proceedings of the sixth international symposium on micro machine and human science*, Ieee, 1995, pp. 39–43.

[15] M. Dorigo, M. Birattari and T. Stutzle, ‘Ant colony optimization,’ *IEEE computational intelligence magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[16] H. Liu, Z. Wen and W. Cai, ‘Fastpso: Towards efficient swarm intelligence algorithm on gpus,’ in *50th International Conference on Parallel Processing*, 2021, pp. 1–10.

[17] J. R. Quinlan, *C4. 5: programs for machine learning.* Elsevier, 2014.

[18] T.-S. Lim, W.-Y. Loh and Y.-S. Shih, ‘A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms,’ *Machine learning*, vol. 40, no. 3, pp. 203–228, 2000.

[19] Y. Freund, R. E. Schapire *et al.*, ‘Experiments with a new boosting algorithm,’ in *icml*, Citeseer, vol. 96, 1996, pp. 148–156.

[20] R. Kohavi and C. Kunz, ‘Option decision trees with majority votes,’ in *ICML*, vol. 97, 1997, pp. 161–169.

[21] R. Maclin and D. Opitz, ‘An empirical evaluation of bagging and boosting,’ *AAAI/IAAI*, vol. 1997, pp. 546–551, 1997.

[22] L. Breiman, ‘Bias, variance, and arcing classifiers,’ Tech. Rep. 460, Statistics Department, University of California, Berkeley ..., Tech. Rep., 1996.

[23] J. H. Friedman, ‘Stochastic gradient boosting,’ *Computational statistics & data analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[24] M. Dorigo, M. Birattari and T. Stutzle, ‘Ant colony optimization,’ *IEEE computational intelligence magazine*, vol. 1, no. 4, pp. 28–39, 2006.

[25] J. H. Friedman, ‘Greedy function approximation: A gradient boosting machine,’ *Annals of statistics*, pp. 1189–1232, 2001.

[26] Y. Shi and R. Eberhart, ‘A modified particle swarm optimizer,’ in *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*, IEEE, 1998, pp. 69–73.

[27] Q. Zhang and W. Wang, ‘A fast algorithm for approximate quantiles in high speed data streams,’ in *19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)*, IEEE, 2007, pp. 29–29.

[28] G. Ke, Q. Meng, T. Finley *et al.*, ‘Lightgbm: A highly efficient gradient boosting decision tree,’ *Advances in neural information processing systems*, vol. 30, 2017.

[29] I. Bida and S. Aouat, ‘A new approach based on bat algorithm for inducing optimal decision trees classifiers,’ in *International Conference Europe Middle East & North Africa Information Systems and Technologies to Support Learning*, Springer, 2018, pp. 631–640.

[30] J. Eggermont, J. N. Kok and W. A. Kosters, ‘Detecting and pruning introns for faster decision tree evolution,’ in *International Conference on Parallel Problem Solving from Nature*, Springer, 2004, pp. 1071–1080.

[31] Y. Shi and R. C. Eberhart, ‘Fuzzy adaptive particle swarm optimization,’ in *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*, IEEE, vol. 1, 2001, pp. 101–106.

[32] Z.-H. Zhan, J. Zhang, Y. Li and H. S.-H. Chung, ‘Adaptive particle swarm optimization,’ *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 6, pp. 1362–1381, 2009.

[33] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush and A. Gulin, ‘Catboost: Unbiased boosting with categorical features,’ *Advances in neural information processing systems*, vol. 31, 2018.

[34] G. Huang, L. Wu, X. Ma *et al.*, ‘Evaluation of catboost method for prediction of reference evapotranspiration in humid regions,’ *Journal of Hydrology*, vol. 574, pp. 1029–1041, 2019.

[35] M. Jünger, G. Reinelt and G. Rinaldi, 'The traveling salesman problem,' *Handbooks in operations research and management science*, vol. 7, pp. 225–330, 1995.

[36] D. Pisinger and P. Toth, 'Knapsack problems,' in *Handbook of combinatorial optimization*, Springer, 1998, pp. 299–428.

[37] J. Kennedy and R. C. Eberhart, 'A discrete binary version of the particle swarm algorithm,' in *1997 IEEE International conference on systems, man, and cybernetics. Computational cybernetics and simulation*, IEEE, vol. 5, 1997, pp. 4104–4108.

[38] K.-P. Wang, L. Huang, C.-G. Zhou and W. Pang, 'Particle swarm optimization for traveling salesman problem,' in *Proceedings of the 2003 international conference on machine learning and cybernetics (IEEE cat. no. 03ex693)*, IEEE, vol. 3, 2003, pp. 1583–1585.

[39] K. Veeramachaneni, L. Osadciw and G. Kamath, 'Probabilistically driven particle swarms for optimization of multi valued discrete problems: Design and analysis,' in *2007 IEEE Swarm Intelligence Symposium*, IEEE, 2007, pp. 141–149.

[40] C. A. Charalambides, *Enumerative combinatorics.* Chapman and Hall/CRC, 2018, p. 42.

[41] P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis, 'Modeling wine preferences by data mining from physicochemical properties,' *Decision support systems*, vol. 47, no. 4, pp. 547–553, 2009.

[42] J. A. Blackard and D. J. Dean, 'Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables,' *Computers and electronics in agriculture*, vol. 24, no. 3, pp. 131–151, 1999.

[43] V. Lohweg, *Banknote authentication data set*, data retrieved from UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets/banknote+authentication#, 2012.

[44] *House sales in king county dataset*, data retrieved from kaggle, https://www.kaggle.com/datasets/harlfoxem/housesalesprediction.

[45] *Medical cost personal datasets*, data retrieved from kaggle, https://www.kaggle.com/datasets/mirichoi0218/insurance.

[46] E. Liang, H. Zhu, X. Jin and I. Stoica, 'Neural packet classification,' in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 256–269.