CSCI-SHU360 Machine Learning Homework 4
Due: Nov 14th 11:59 PM, 2024 (GMT+8)
100 points

Instructions

- **Collaboration policy:** Homework must be done individually, except where otherwise noted in the assignments. "Individually" means each student must hand in their own answers, and you must write and use your own code in the programming parts of the assignment. It is acceptable for you to collaborate in figuring out answers and to help each other solve the problems, and you must list the names of students you discussed this with. We will assume that, as participants in an undergraduate course, you will be taking the responsibility to make sure you personally understand the solution to any work arising from such collaboration.
- **Online submission:** You must submit your solutions online on the course **Gradescope** site (you can find a link on the course Brightspace site). You need to submit (1) a PDF that contains the solutions to all questions to the Gradescope **HW4 Paperwork** assignment (**including the questions asked in the programming problems**), (2) `x.py` or `x.ipynb` files for the programming questions to the Gradescope **HW4 Code Files** assignment. We recommend that you type the solution (e.g., using LaTeX or Word), but we will accept scanned/pictured solutions as well (clarity matters).
- **Generative AI Policy:** You are free to use any generative AI, but you are required to document the usage: which AI do you use, and what's the query to the AI. You are responsible for checking the correctness.
- **Late Policy: No late submission is allowed** as we give extra time and reduce the number of questions.

Before you start: this homework only has programming problems. You should still have all questions answered in the write-up pdf. Also note that some sub-problems are still essentially math problems and you need to show detailed derivations.

Problems 1 and 2 are two parts of one problem, so we suggest you read the description of both problems in tandem.

We will/have introduced random forests and gradient boosting decision trees (GBDTs) in class, and we also include a detailed description here if you want to start early or use it as a review for the lecture. You can also check the slides in advance if you want.

This homework could be challenging and hope the following tips help:

- Understanding of the gradient boosting tree concept. In particular, we use every single tree to compute a "gradient step", and the sum of the gradient steps gives us the final predictions.
- Understanding of the python notebook we provide. The code we provide aims to share implementations between the random forests and the GBDTs. Try to think about how different parts of the code could be re-utilized by the two models.
- Debugging your code: always try to debug a small case. For example, use very few data points and build a tree with a depth of 2 or 3. Then, you can look at all the decision rules and data point assignments and check if they are reasonable.

1 Programming Problem: Random Forests [40 points]

Random forests (RF) build an ensemble of trees independently. It uses bootstrap sampling (sampling with replacement, as discussed in class) to randomly generate B datasets from the original training dataset, each with the same size as the original one but might contain some duplicated (or missing) samples. Each sample has a multiplicity which is greater than or equal to zero. In your python implementation, you can use numpy.random.choice for the sampling procedure.

The RF procedure independently trains B decision tree models $\{f_b(\cdot;\theta_b)\}_{b=1}^B$ on these B datasets and these are done independently. To train each tree, we start from a root node with all the assigned data, and recursively splitting the node and its data into two child nodes, by a decision rule on a feature dimension (thresholding in particular). When we're all done, RFs produces predictions by averaging the outputs of all the B models as follows:

$$\hat{y}_i = \frac{1}{B}\sum_{b=1}^{B} f_b(x_i;\theta_b). \tag{1}$$

The optimization problem for training each tree in RF is

$$\min_{\theta_b} \sum_{i=1}^{n} \ell(y_i, \hat{y}_i) + \Omega(\theta_b), \tag{2}$$

where $\hat{y}_i$ is the prediction produced by tree-b $f_b(\cdot;\theta_b)$ for data point x_i, $\ell(\cdot,\cdot)$ is a loss function (detailed in Problem 2.5.3), and $\Omega(\theta_b)$ is a regularizer applied to the parameters θ_b of model-b (that is, $\Omega(\theta_b)$ measures the complexity of model-k). Most descriptions of ensemble learning in Problem 2.1 of the homework (below) can be also applied to RF, such as the definitions of $f_k(\cdot;\theta_k)$ and θ_k, except Eq. (3) and Eq. (4).

Different methods can be used to find the decision rule on each node during the optimization of a single tree. A core difference between random forests and GBDTs (which we will describe in Problem 2) is the tree growing methods. Specifically, in the case of GBDT, we use the standard greedy tree-splitting algorithm; in the case of random forests, we greedily learn each tree using a bootstrapped data sample and random feature selection as described in class. That is, the key difference is the data that is being used (always original data in the case of GBDT or bootstrap sample for each tree in the case of RFs), and in the case of RFs we choose a random subset of features each time we grow a node. The underlying algorithm, however, is very similar. Therefore, to facilitate code reuse between this and the next problem, and also to make more fair the comparison between RFs and GBDTs, we ask you to use the same code base between this and the next problem (detailed in Problem 2.4 below).

Each tree in the RF method is like the first tree in GBDT, as the RF method does not consider any previously produced trees when it grows a new tree (the trees are independent with RFs). With RFs, we simply start with $\hat{y}_i^0$. You need to notice this fact when re-using the code from GBDT, because G_j and H_j for tree-k in RF only depend on $\hat{y}_i^0$, not $\hat{y}_i^{k-1}$. Instructions 2-5 in Problem 2.5, however, can still be applied to RF tree building here.

In this problem, you will implement RFs for both regression and binary classification problems. Please read Problem 2.1, 2.4, and 2.5 below before you start.

1. *[20 points]* Implement RF for regression task, and test its performance on Boston house price dataset[1] used in Homework 2. Report the training and test RMSE. How is the performance of RF compared to least square regression and ridge regression?

2. *[20 points]* Implement RF for binary classification task, and test its performance on Credit-g dataset. It is a dataset classifying people described by 20 attributes as good or bad credit risks. The full description of the attributes can be found at `https://www.openml.org/d/31`. Report the training and test accuracy. Try your implementation on breast cancer diagnostic dataset[2], and report the training and test accuracy.

[1] `https://www.kaggle.com/c/boston-housing`
[2] `https://www.kaggle.com/uciml/breast-cancer-wisconsin-data`

2 Programming Problem: Gradient Boosting Decision Trees [60 points]

2.1 Problem of Ensemble Learning in GBDTs

Gradient Boosting Decision Trees (GBDT) is a class of methods that use an ensemble of K models (decision trees) $\{f_k(\cdot;\theta_k)\}_{k=1}^K$. It produces predictions by adding together the outputs of the K models as follows:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i;\theta_k). \tag{3}$$

The resulting $\hat{y}$ can be used for the predicted response for regression problems, or can correspond to the class logits (i.e., the inputs to a logistic or softmax function to generate class probabilities) when used for classification problems.

GBDT has been widely used in a variety of real applications and industrial fields with great success, due to GBDT's training efficiency (especially compared to deep neural networks), and more importantly, its interpretability, since decision trees and the decision rules on them are much more transparent for humans to understand and explain than high-dimensional model weights and artificial non-linearly aggregated features produced by complicated deep neural networks. In addition, GBDT methods have been the winner of a large number of machine learning competitions, for example, in many Kaggle competitions[3], where people can earn prizes and money from winning simply by using GBDT (via standard packages XGBoost or LightGBM below). Furthermore, GBDT has empirically shown itself to be more robust to noise and data imbalance issues on different types of data and tasks than other methods. There are many GBDT packages that optimize the efficiency of almost every step of GBDT and make the algorithm extremely fast and easy to use, for example, XGBoost[4] and LightGBM[5]. In the problem below, you are not allowed to use any of these packages in your code, but you need to implement GBDT from scratch. GBDT is an extremely useful state-of-the-art ML tool. Given the success of deep neural networks, it is heartening that a different class of methods often performs so well. In the below, you will learn to implement it step by step, by following the tutorial below.

The optimization problem for training an ensemble of models is

$$\min_{\{\theta_k\}_{k=1}^K} \sum_{i=1}^{n} \ell(y_i,\hat{y}_i) + \sum_{k=1}^{K} \Omega(\theta_k), \tag{4}$$

where θ_k is the parameters for the k^{th} model, and where $\hat{y}_i$ is the prediction produced by GBDT for data point x_i, $\ell(\cdot,\cdot)$ is a loss function (detailed definition of losses are given in Problem 2.5.3), and $\Omega(\theta_k)$ is a regularizer applied to the parameters θ_k of model-k (that is, $\Omega(\theta_k)$ measures the complexity of model-k).

In GBDT, each model $f_k(\cdot;\theta_k)$ is a decision tree that assigns each data point x to one of its leaf nodes $q_k(x) \in L_k$, where L_k is the set of all the leaf nodes in the k^{th} tree. Each leaf node $j \in L_k$ has a weight w_j^k, and $(w_1^k, w_2^k, \ldots, w_{|L_k|}^k) = w^k \in \mathbb{R}^{|L_k|}$ is a $|L_k|$-dim vector storing the weights of all the leaf nodes. We may write the decision tree function as follows

$$f_k(x) = w^k_{q_k(x)}, \tag{5}$$

where $q_k(\cdot) : \mathbb{R}^d \mapsto L_k$ represents the decision process of the k^{th} decision tree. That is, $q_k(x)$ assigns each data x to a leaf node j of the k^{th} tree; it is comprised of the decision rules on all the non-leaf nodes as its learnable parameters. In particular, on each non-leaf node $j \in N_k$ (N_k is the set of all the non-leaf nodes on tree-k), the decision rule is defined by a choice of feature dimension $p_j \in [m]$ and a choice of threshold τ_j leading to the following rule: when $x_{p_j} < \tau_j$, x is assigned to the left child node of j, otherwise x is assigned to the right child node.

Therefore, to define a tree $f_k(\cdot)$, we need to determine the structure of the tree $\mathcal{T} \triangleq (N_k \cup L_k, E)$ (E is the set of tree edges), the feature dimension p_j and threshold τ_j associated with each non-leaf node $j \in N_k$,

[3] `https://www.kaggle.com/competitions`
[4] `https://github.com/dmlc/xgboost`
[5] `https://github.com/Microsoft/LightGBM`

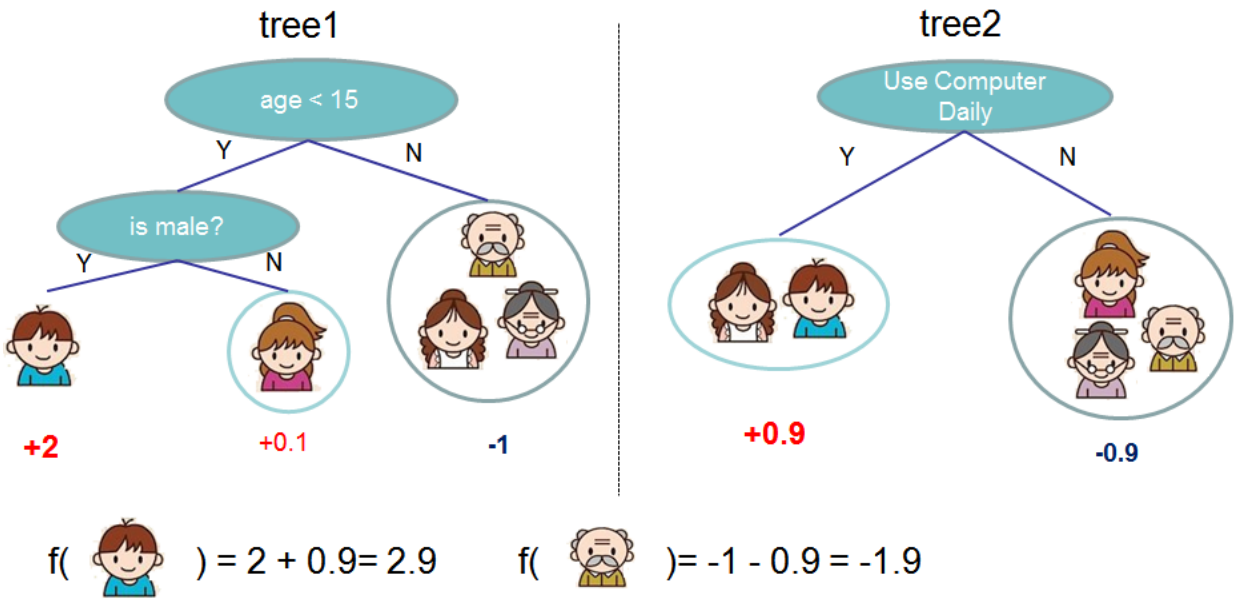

Figure 1: An example of GBDT: Does the person like computer games?

and the weight w_j^k associated with each leaf node $j \in L_k$. These comprise the learnable parameters θ_k of $f_k(\cdot;\theta_k)$, i.e.,

$$\theta_k = \left\{\mathcal{T}, w^k, \{(p_j, \tau_j)\}_{j\in N_k}\right\}. \tag{6}$$

To define an ensemble of multiple trees, we also need to know the number of trees K.

We cannot directly apply gradient descent to learn the above parameters of GBDT because: (1) some of the above variables are discrete and some could have an exponential number of choices, including for example, the number of trees, the structure of each tree, the feature dimension choice associated with each non-leaf node, the weights at the leaf nodes; and 2) the overall decision tree process is not differentiable meaning straightforward naive gradient descent seems inapplicable.

2.2 Overview of the GBDT algorithm

The basic idea of GBDT training is additive training, or boosting. As mentioned above, Boosting is a meta-algorithm that trains multiple models one after another, and in the end combines additively to produce a prediction. Boosting often aims to convert multiple weak learners (which might be only slightly better than a random guess) into a strong learner that can achieve error arbitrarily close to zero. Boosting has many forms and instantiations, including AdaBoost [6], random forests [5, 2], gradient boosting [3, 4], etc. Note that Bagging [1] is not boosting since there is no interdependence between the trainings of the different models, rather each model in Bagging is trained on a separate bootstrap data sample.

GBDT training shares ideas similar to coordinate descent in that only one part of the model is optimized at a time, while the other parts are fixed. In the coordinate descent algorithm (you implemented it for Lasso), each outer loop iteration requires one pass of all the feature dimensions. In each iteration of its inner loop, it starts from the first dimension, and optimizes only one dimension of the weight vector by fixing all the other dimensions and conditioning on all the other dimensions. Each tree in GBDT is analogous to a dimension in coordinate descent, but the optimization process is different. GBDT starts from the first tree, and optimizes only one tree per time by fixing all the previous trees and we condition on all the previously produced trees. One core difference GBDT training has from coordinate descent is that GBDT training does not have the outer loop associated with coordinate descent, i.e., it only does one pass over all the trees. If it was coordinate descent, we would optimize each coordinate only once in succession.

In particular, we start from one tree, and only optimize one tree at a time conditioned on all previously produced trees. This, therefore, is a greedy strategy as we spoke about in class. After the training of one tree

finished, we add this new tree to our growing ensemble and then repeat the above process. The algorithm stops when we have added t_{max} trees to the ensemble.

In the optimization of each tree, we start from a root node, find the best decision rule (a feature dimension and a threshold) and split the node into two children, go to each of the children, and recursively find the best decision rule on each of the child nodes, and continue until some stopping criterion is fulfilled (as will be explained very shortly below).

In the following, we will first elaborate how to add trees one after the other, and then provide details regarding how to optimize a single tree based on a set of previous trees (which might be empty, and so this also explains how to start with the first tree).

2.3 Growing the forest: How to add a new tree?

Assume that there will be K trees in the end. Therefore, we will get a sequence of (partially) aggregated predictions $\{\hat{y}_i^k\}_{k=1}^K$, from the K trees as follows:

$$\begin{aligned}
\hat{y}_i^0 &= 0, \\
\hat{y}_i^1 &= f_1(x_i) = \hat{y}_i^0 + f_1(x_i), \\
\hat{y}_i^2 &= f_1(x_i) + f_2(x_i) = \hat{y}_i^1 + f_2(x_i), \\
&\cdots \\
\hat{y}_i^k &= \sum_{k'=1}^{k} f_{k'}(x_i) = \hat{y}_i^{k-1} + f_k(x_i), \\
&\cdots \\
\hat{y}_i^K &= \sum_{k=1}^{K} f_k(x_i) = \hat{y}_i^{K-1} + f_K(x_i).
\end{aligned}$$

According to Eq. (4), fixing all the previous $k-1$ trees, the objective used to optimize tree-k is

$$F_k(\theta_k) = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i^k) + \Omega(\theta_k) = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i^{k-1} + f_k(x_i)) + \Omega(\theta_k), \tag{7}$$

Let's simplify the first term using Taylor's expansion, ignoring higher-order terms:

$$f(x+\Delta) \approx f(x) + f^{'}(x)\Delta + \frac{1}{2} f^{''}\Delta^2. \tag{8}$$

After applying Taylor's expansion to $\ell(y_i, \hat{y}_i^{k-1} + f_k(x_i))$, we have

$$\ell(y_i, \hat{y}_i^{k-1} + f_k(x_i)) \approx \ell(y_i, \hat{y}_i^{k-1}) + g_i f_k(x_i) + \frac{1}{2} h_i f_k^2(x_i), \tag{9}$$

where g_i and h_i denote the first-order and second-order derivatives of $\ell(y_i, \hat{y}_i^{k-1})$ w.r.t. $\hat{y}_i^{k-1}$, i.e.,

$$g_i \triangleq \frac{\partial \ell(y_i, \hat{y}_i^{k-1})}{\partial \hat{y}_i^{k-1}}, \quad h_i \triangleq \frac{\partial^2 \ell(y_i, \hat{y}_i^{k-1})}{(\partial \hat{y}_i^{k-1})^2} = \frac{\partial g_i}{\partial \hat{y}_i^{k-1}}. \tag{10}$$

The second term $\Omega(\theta_k)$ in Eq. (7) is a regularization term aiming to penalize the degree of complexity of tree-k. It depends on the number of leaf nodes, and the L2 regularization of w^k. With GBDTs, it is defined as:

$$\Omega(\theta_k) \triangleq \gamma |L_k| + \frac{1}{2}\lambda \|w^k\|_2^2. \tag{11}$$

We plugin Eq. (9) and Eq. (11) into Eq. (7), and after ignoring constants, we get

$$\begin{aligned} F_k(\theta_k) + \text{const.} &\approx \sum_{i=1}^{n} \left[g_i f_k(x_i) + \frac{1}{2} h_i \cdot f_k^2(x_i) \right] + \gamma |L_k| + \frac{1}{2} \lambda \|w^k\|_2^2 \\ &= \sum_{i=1}^{n} \left[g_i w^k_{q_k(x_i)} + \frac{1}{2} h_i \cdot (w^k_{q_k(x_i)})^2 \right] + \gamma |L_k| + \frac{1}{2} \lambda \|w^k\|_2^2 \\ &= \sum_{j \in L_k} \left[\left(\sum_{i \in I_j} g_i \right) w_j^k + \frac{1}{2} \left(\lambda + \sum_{i \in I_j} h_i \right) (w_j^k)^2 \right] + \gamma |L_k|, \end{aligned} \tag{12}$$

where I_j represents the set of all the data points assigned to leaf $j \in L_k$, i.e., $I_j \triangleq \{i \in [n] : q_k(x_i) = j\}$. Eq. (12) gives us the objective of optimizing a tree conditioned on all the previously achieved trees.

2.4 Growing a tree: How to optimize a single tree?

Now we can start to optimize a single tree $f_k(\cdot; \theta_k)$. Look at the objective function in Eq. (12): it is a sum of $|L_k|$ independent simple scalar quadratic functions of w_j^k for all the $j \in L_k$! How to minimize a quadratic function? This we know is easy, and similar to least square regression, the solution has a nice closed form. Hence, w_j^k minimizing $F_k(\theta_k)$ is

$$w_j^k = -\frac{G_j}{H_j + \lambda}, \quad G_j \triangleq \sum_{i \in I_j} g_i, \quad H_j \triangleq \sum_{i \in I_j} h_i. \tag{13}$$

We can plug the above optimal w_j^k into Eq. (12) and obtain an updated objective

$$F_k \left(\theta_k; \left\{ w_j^k = -\frac{G_j}{H_j + \lambda} \right\}_{j \in L_k} \right) = \gamma |L_k| - \frac{1}{2} \sum_{j \in L_k} \frac{G_j^2}{H_j + \lambda} \tag{14}$$

However, there are still two groups of unknown parameters in θ_k, which are the tree structure $\mathcal{T}$ and the decision rules $\{(p_j, \tau_j)\}_{j \in N_k}$. In the following, we will elaborate how to learn these parameters by additive training of a single tree.

We will start from the root node and determine the associated decision rule (p_j, τ_j); this rule should minimize the updated objective in Eq. (14), where L_k contains the left and right child of the root node. Then, the same process of determining (p_j, τ_j) will be recursively applied to the left and right nodes, until a stopping criteria (as described below) is fulfilled.

Firstly, let's determine the number of all the possible choices of (p_j, τ_j) for node $j \in N_k$. Since there are m features, we have m different choices for p_j. For each feature dimension $p_j \in [m]$, we can sort all the $n_j \triangleq |I_j|$ data points assigned to node j by their feature values on dimension p_j, so there are at most n_j choices [6] of thresholds τ_j when we based them on the n training data points. In particular, If the sequence of sorted feature values is $(v_1, v_2, v_3, \ldots, v_{n_j})$, the t^{th} choice of threshold can be any value between v_t and v_{t+1}. We usually use the middle point $\frac{1}{2}(v_t + v_{t+1})$ as the t^{th} candidate of threshold. Therefore, there are $m \times n_j$ choices for (p_j, τ_j), i.e., each of the m choices for p_j has n_j choices of threshold τ_j.

For each candidate decision rule (p_j, τ_j), we can compute the improvement it brings to the objective Eq. (14). Before splitting node j to a left child $j(L)$ and a right child $j(R)$, the objective is

$$F_k \left(\theta_k; \left\{ w_j^k \right\}_{j \in L_k} \right) = \gamma |L_k| - \frac{1}{2} \sum_{j' \in L_k \setminus j} \frac{G_{j'}^2}{H_{j'} + \lambda} - \frac{1}{2} \frac{G_j^2}{H_j + \lambda} \tag{15}$$

After splitting, the leaf nodes change to $j(L)$ and $j(R)$, and the objective becomes

$$F_k \left(\theta_k; w^k_{j(L)}, w^k_{j(R)}, \left\{ w^k_{j'} \right\}_{j' \in L_k \setminus j} \right) = \gamma (|L_k| + 1) - \frac{1}{2} \sum_{j' \in L_k \setminus j} \frac{G_{j'}^2}{H_{j'} + \lambda} - \frac{1}{2} \frac{G^2_{j(L)}}{H_{j(L)} + \lambda} - \frac{1}{2} \frac{G^2_{j(R)}}{H_{j(R)} + \lambda} \tag{16}$$

[6] It can be less than n_j if there are duplicated feature values on different data points

Hence, the improvement (we usually call it the "gain") is

$$F_k\left(\theta_k; \{w_j^k\}_{j\in L_k}\right) - F_k\left(\theta_k; w_{j(L)}^k, w_{j(R)}^k, \{w_{j'}^k\}_{j'\in L_k\setminus j}\right) \tag{17}$$

$$= \frac{1}{2}\left[\frac{G_{j(L)}^2}{H_{j(L)}+\lambda} + \frac{G_{j(R)}^2}{H_{j(R)}+\lambda} - \frac{G_j^2}{H_j+\lambda}\right] - \gamma \tag{18}$$

$$= \frac{1}{2}\left[\frac{G_{j(L)}^2}{H_{j(L)}+\lambda} + \frac{G_{j(R)}^2}{H_{j(R)}+\lambda} - \frac{(G_{j(L)}+G_{j(R)})^2}{H_{j(L)}+H_{j(R)}+\lambda}\right] - \gamma. \tag{19}$$

Therefore, the best decision rule (p_j, τ_j) on node $j \in N_k$ is the one (out of the $m \times n$ possible rules) maximizing the gain, which corresponds to the decision rule that minimizes the updated objective in Eq. (14). That is, we wish to perform the following optimization:

$$\max_{(p_j,\tau_j)} \frac{1}{2}\left[\frac{G_{j(L)}^2}{H_{j(L)}+\lambda} + \frac{G_{j(R)}^2}{H_{j(R)}+\lambda} - \frac{(G_{j(L)}+G_{j(R)})^2}{H_{j(L)}+H_{j(R)}+\lambda}\right] - \gamma. \tag{20}$$

We start from the root node, apply the above criterion to find the best decision rule, split the root into two child nodes, and recursively apply the above criterion to find the decision rules on the child nodes, the grandchildren, and so on. We stop splitting according to a stopping criterion is satisfied. In particular, we stop to split a node if either of the following events happens:

1. the tree has reached a maximal depth d_{max};
2. the improvement achieved by the best decision rule for the node (Eq. (20)) goes negative (or is still positive but falls below a small positive threshold, in the following experiments, you can try this, but please report results based on the "goes negative" criterion);
3. The number of samples assigned to the node is less than n_{min}.

2.5 Details of Practical Implementation

1. **Learning rate η:** You might notice that the tree growing in GBDT is a greedy process. In practice, to avoid overfitting on a single tree, and to give more chances to new trees, we will make the process less greedy. In particular, we usually assign a weight $0 \le \eta \le 1$ to each newly added tree when aggregating its output with the outputs of previously added trees. Hence, the sequence at the beginning of Problem 2.3 becomes

$$\begin{aligned}
&\hat{y}_i^0 = 0,\\
&\hat{y}_i^1 = \eta f_1(x_i) = \hat{y}_i^0 + \eta f_1(x_i),\\
&\hat{y}_i^2 = \eta f_1(x_i) + \eta f_2(x_i) = \hat{y}_i^1 + \eta f_2(x_i),\\
&\qquad\cdots\\
&\hat{y}_i^k = \eta \sum_{k'=1}^{k} f_{k'}(x_i) = \hat{y}_i^{k-1} + \eta f_k(x_i),\\
&\qquad\cdots\\
&\hat{y}_i^K = \eta \sum_{k=1}^{K} f_k(x_i) = \hat{y}_i^{K-1} + \eta f_K(x_i).
\end{aligned}$$

   Note that this change must be applied in both training and during testing/inference. $0 \le \eta \le 1$ is usually called the "learning rate" of GBDT, but it is not exactly the same as the variable we usually call the learning rate in gradient descent.

2. **Initial prediction $\hat{y}_i^0$:** GBDT does not have bias term b like linear model $y = wx + b$. Fortunately, $\hat{y}_i^0$ plays a similar role as b. Hence, instead of starting from $\hat{y}_i^0 = 0$, we start from $\hat{y}_i^0 = \frac{1}{n}\sum_{i=1}^n y_i$, i.e., the average of ground truth on the training set. For classification, it is also fine to use this initialization (the average of lots of 1s and 0s), but do not forget to transfer the data type of label from "int" to "float" when computing the average in this case.

3. **Choices of loss function $\ell(\cdot,\cdot)$:** $\ell(\cdot,\cdot)$ is a sample-wise loss. In the experiments, you should use least square loss $\ell(y,\hat{y}) = (y-\hat{y})^2$ for regression problems. For binary classification problems, we use one-hot (0/1) encoding of labels y (y is either 0 or 1), and logistic regression (the GBDT output $\hat{y}$ is the logit in this case, which is a real number and the input to logistic function producing class probability), i.e.,

$$\ell(y,\hat{y}) = y\log(1+\exp(-\hat{y})) + (1-y)\log(1+\exp(\hat{y})). \tag{21}$$

   The prediction of binary logistic regression, which is the class probabilities, is

$$\Pr(class=1) = \frac{1}{1+\exp(-\hat{y})}, \quad \Pr(class=0) = 1 - \Pr(class=1). \tag{22}$$

   To produce a one-hot (0/1) prediction, we apply a threshold of 0.5 to the probability, i.e.,

$$class = \begin{cases} 1, & \Pr(class=1) > 0.5 \\ 0, & \Pr(class=1) \leq 0.5 \end{cases} \tag{23}$$

4. **Hyper-parameters:** There are six hyper-parameters in GBDT, i.e., λ and γ in regularization $\Omega(\cdot)$, d_{max} and n_{min} in stopping criterion for optimizing single tree, maximal number of trees t_{max} in stopping criterion for growing forests, and learning rate η. We will not give you exact values for these hyper-parameters, since tuning them is an important skill in machine learning. Instead, we will give you ranges of them for you to tune. Note larger d_{max} and t_{max} require more computations. Their ranges are: $\lambda \in [0,10]$, $\gamma \in [0,1]$, $d_{max} \in [2,10]$, $n_{min} \in [1,50]$, $t_{max} \in [5,50]$, $\eta \in [0.1,1.0]$.

   In RFs, we do not have the learning rate, but there is another hyper-parameter, which is the size m' of the random subset of features, from which you need to find the best feature and the associated decision rule for a node. You can use $m' \in [0.2m, 0.5m]$.

5. **Stopping criteria:** There are two types of stopping criteria needed to be used in GBDT/RFs training: 1) we stop to add new trees once we get t_{max} trees; and 2) we stop to grow a single tree once either of the three criteria given at the end of Problem 2.4 fulfills.

6. **Acceleration:** We encourage you to apply different acceleration methods after you make sure the code works correctly. You can use multiprocessing for acceleration, and it is effective. However, do not increase the number of threads to be too large. It will make it even slower. You can also try numba (a python compiler) with care.

7. **Code template:** We provide you a template of RF and GBDT implementation in a ipython notebook "GBDT.ipynb". You can change the existing code in it for your convenience. The notebook also includes the code of loading the three datasets, so you do not need to download them manually. **Please note that most code for regression and classification can be shared, and the only difference is the loss function (pred(), g() and h() in notebook), so you do not need to implement them twice: their difference is just a couple of lines and a loss function option argument for your GBDT class.**

2.6 Questions

1. *[4 points]* What is the computational complexity of optimizing a tree of depth d in terms of m and n?

2. *[4 points]* What operation requires the most expensive computation in GBDT training? Can you suggest a method to improve the efficiency (please do not suggest parallel or distributed computing here since we will discuss it in the next question)? Please give a short description of your method.

3. *[8 points]* Which parts of GBDT training can be computed in parallel? Briefly describe your solution, and use it in your implementation. (Hint: you might need to use "from multiprocessing import Pool" and "from functools import partial". We also talked about multiprocessing in the recitation session.)

4. *[20 points]* Implement GBDT for the regression task, and test its performance on Boston house price dataset used in Homework 2. Report the training and test RMSE. How is the performance of GBDT compared to least square regression and ridge regression?

5. *[20 points]* Implement GBDT for the binary classification task, and test its performance on Credit-g dataset. Report the training and test accuracy. Try your implementation on the breast cancer diagnostic dataset, and report the training and test accuracy.

6. *[4 points]* According to the results on the three experiments, how is the performance of random forests compared to GBDT? Can you give some explanations?

References

[1] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[2] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[3] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38:367–378, 1999.

[4] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[5] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282, 1995.

[6] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.