EE511 Advanced Intro. to Machine Learning **University of Washington**
Spring 2020 **Dept. of Electrical & Computer Engineering**

Homework 4. Due **Monday, March 8th, 11:59pm** Electronically

Prof: J. Bilmes <bilmes@uw.edu> *Friday, Feb 26th, 2020*
TA: Lilly Kumari <lkumari@uw.edu>

There are two due dates associated with this assignment. The preliminary read-through due date of this assignment is **due Tuesday, March 2nd, 5:30pm**. See https://canvas.uw.edu/courses/1431528/assignments/6107236. **No lates are accepted for this preliminary due date.¡**
The final and complete assignment is **due Monday, March 8th, at 11:59pm** via https://canvas.uw.edu/courses/1431528/assignments/6107279.

General Instructions.

This homework consists of two parts, a write-up part that needs to be turned in using a single pdf file, and also a programming part that needs to be turned in using a zipped Jupyter notebook file (or files). A Jupyter notebook file has extenion `.ipynb` and a zip file has extension `.zip` and should contain one or more `.ipynb` files.

Doing your homework by hand and then converting to a PDF file (by say taking high quality photos using a digital camera and then converting that to a PDF file) is fine, as there are many jpg to pdf converters on the web. Alternatively, you are welcome to use Latex (great for math and equations), Microsoft Word, and Google Docs, or hand-written paper, as long as the final submitted format is a single pdf.

For the plots requested in the programming session, you can either save them as pictures and insert them manually into the writeup, or directly export the completed jupyter notebook to a pdf file (in jupyter notebook, "File→Download as→PDF via LaTex") and copy it in to your writeup.

Some of the problems below might require that you look at some of the lecture slides at our web page (https://canvas.uw.edu/courses/1431528).

Note that the due dates and times are often in the evenings.

As mentioned above, for the programming problems, you need to submit your code (written in python as a Jupyter notebook) and the answers to the non-coding questions should also be included in the pdf write-up. Your code answers must to be in python, no other language is accepted.

Neatness and clarity count! : Answers to your questions must be clearly indicated in all cases. Not only correctness, but clarity and completeness is necessary to receive full credit. Justify you answers. A correct answer does not guarantee full credit and a wrong answer does not guarantee poor credit, hence show all work and justify each step, thinking "clarity" and "neatness" along the way. If we can't understand your answer, or if your answers are not well and neatly organized, you will not receive full credit.

All homework is due electronically via the link https://canvas.uw.edu/courses/1431528/assignments. This means that on canvas you turn in two files: (1) **a pdf file** with answers to the writeup questions, and (2) **a zip file with python code (in jupyter notebook files)**. **Please do not submit any fewer or any more than these two files.**

Problem 1. PCA via Successive Deflation [30 points] (Adapted from Murphy Exercise 12.7)

Given a design matrix $\mathbf{X}$ (i.e., $n \times m$ matrix of n rows and m columns, and suppose $n \geq m$ and that the matrix has rank m), every row of $\mathbf{X}$ corresponds to the feature vector of a data point. We assume that the matrix is zero mean (meaning the mean of every column is zero). Consider the transpose, i.e. $\mathbf{X}^T$, where $\mathbf{X}^T = [x_1; \ldots; x_n]$, and x_i is a column vector representing a data point. Naively, computing PCA of $\mathbf{X}$

involves solving eigen values/vectors of the covariance matrix $\mathbf{C} = \frac{1}{n}\mathbf{X}^T\mathbf{X}$ (since it is zero mean), which can be intractable for very large m.

Suppose we order the eigenvectors of $\mathbf{C}$ by their eigenvalues (largest first), and let $v_1, v_2, \ldots v_k$ be the first k eigenvectors (normalized). Thanks to orthonormality, these satisfy

$$v_j^T v_k = \begin{cases} 0 & \text{if } j \neq k \\ 1 & \text{if } j = k \end{cases}$$

Note that v_1 is the first principal eigenvector of $\mathbf{C}$ (the eigenvector with the largest eigenvalue), and as such it satisfies $\mathbf{C}v_1 = \lambda_1 v_1$. Now define $\tilde{x}_i$ as the orthogonal projection of x_i onto the space orthogonal to v_1:

$$\tilde{x}_i = (\mathbf{I} - v_1 v_1^T)x_i$$

Define $\tilde{\mathbf{X}}^T = [\tilde{x}_1; \ldots; \tilde{x}_n]$ as the **deflated matrix** of rank $m-1$, which is obtained by removing from the m-dimensional data the component that lies in the direction of the first principal eigenvector:

$$\tilde{\mathbf{X}}^T = (\mathbf{I} - v_1 v_1^T)\mathbf{X}^T$$

Problem 1(a). [10 points] Show that the covariance of the deflated matrix,

$$\tilde{\mathbf{C}} = \frac{1}{n}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$$

can be computed by

$$\tilde{\mathbf{C}} = \frac{1}{n}\mathbf{X}^T\mathbf{X} - \lambda_1 v_1 v_1^T$$

Problem 1(b). [5 points] Show that for $j \neq 1$, if v_j is a principal eigenvector of $\mathbf{C}$ with corresponding eigenvalue λ_j (that is, $\mathbf{C}v_j = \lambda_j v_j$), then v_j is also a principal eigenvector of $\tilde{\mathbf{C}}$ with the same eigenvalue λ_j.

Problem 1(c). [5 points] Let u be the first principal eigenvector of $\tilde{\mathbf{C}}$. Explain why $u = v_2$. (You may assume u has unit norm.)

Problem 1(d). [5 points] There are efficient methods to compute the leading eigen value/vector of a positive-definite matrix. One simple approach is the power iteration method, where given the matrix $\mathbf{C}$, we take a (non-zero) guess of the eigenvector u_0, and iteratively multiply $\mathbf{C}$ with the current guess:

$$u_1 = \mathbf{C}u_0 \tag{1}$$
$$u_2 = \mathbf{C}u_1 = \mathbf{C}^2 u_0 \tag{2}$$
$$\ldots \tag{3}$$
$$u_k = \mathbf{C}u_{k-1} = \mathbf{C}^k u_0 \tag{4}$$

For sufficiently many iterations, we can get u_k as a good approximation of the leading eigenvector. Intuitively explain why this works. What's the computational complexity (suppose we do k iterations)? Also, explain in terms of Eigenvectors/Eigenvalues why a better approach when you implement this would have each iteration do:

$$u_i = \frac{\mathbf{C}u_{i-1}}{\|\mathbf{C}u_{i-1}\|_2} \tag{5}$$

Problem 1(e). [5 points] Now let's combine power iteration with our successive deflation method. Let's call the power iteration method f, which has the form $[\lambda, u] = f(\mathbf{C}, k)$. Write down the python/numpy code for finding the first K principal basis vectors of $\mathbf{X}$ that only uses f and simple vector arithmetic, and test that it is working. What's the computational complexity?

(Hint: This should be a simple iterative routine that takes only a few lines to write. The input is $\mathbf{C}, K, k$

and the function f, where k is the number of iterations of the power iteration method to run, and K is the number of Eigenvectors to compute. The output should be v_j and λ_j for $j \in [1 : K]$.)

Problem 2. Locality Sensitive Hashing (LSH) [20 points]

Locality sensitive hashing refers to a special property of hash functions and is closely related to *approximate* nearest neighbor search (NNS), i.e., we find close points to the query instead of exactly the nearest neighbor.

For simplicity, suppose the design matrix $\mathbf{X}$ (which is $n \times m$) consists of only binary features. This means that every data point (i.e., every row of the design matrix) has the form $x_i \in \{0,1\}^m$ (i.e., is a binary vector). Define $d(x_i, x_j)$ as the Hamming distance between two data points x_i and x_j, i.e. $d(x_i, x_j) = \sum_{a \in \{0,1,\ldots,m-1\}} |x_i[a] - x_j[a]|$. The Hamming distance simply counts the number of positions where the two binary vectors differ. Also, the Hamming distance is a proper distance **metric** (see https://en.wikipedia.org/wiki/Metric_(mathematics) for the full definition of a distance metric).

Problem 2(a). [5 points] Imagine you have the following magical oracle. Given a query point $q \in \{0,1\}^m$, and two parameters $r > 0$ and $c \geq 1$,

1. If $\exists x \in \mathbf{X}$ such that $d(x, q) \leq r$, the oracle returns to you some point $x' \in \mathbf{X}$ such that $d(x', q) \leq cr$.
2. If $\nexists x \in \mathbf{X}$ such that $d(x, q) \leq cr$, the oracle returns to you nothing.
3. Otherwise ($\exists x \in \mathbf{X}$ such that $d(x, q) \leq cr$ but $\nexists x \in \mathbf{X}$ such that $d(x, q) \leq r$), the oracle is not stable, and can either return to you some point x' such that $d(x', q) \leq cr$ or return to you nothing.

Suppose you want to find the exact nearest neighbor point in $\mathbf{X}$ of the query point q. Calling the magical oracle as few times as possible, how do you appropriately set the values r and c in order to get back the nearest neighbor of q?

Problem 2(b). [2 points] Unfortunately the magical oracle is not free. Suppose we set r to be slightly larger than the distance to the nearest neighbor, i.e. $r = \min_{x \in \mathbf{X}} d(x, q) + \epsilon$, if we set a very large c, the oracle may return to us any data point, which is cheap but not useful. However, if we set a small c, the oracle becomes expensive and returns to us a point with distance at most cr, which may serve as a good approximation to the true nearest neighbor. By setting values of c, we are controlling the trade-off between the quality of approximate nearest neighbor and the oracle's running time. We will then show how to implement the oracle using locality sensitive hashing functions.

Consider a family of hash functions H, where each function h is associated with an integer a from $\{0, 1, \ldots, m-1\}$, and given the input x_i, $h(x_i) = x_i[a]$. This hash function is very simple, and merely returns coordinate a of the data point x_i.

Suppose we uniformly at random pick one of the m hash functions h from H. For two input data points x_i and x_j, if $d(x_i, x_j) \leq r$ (with $r > 0$), what's a lower bound for the probability that h maps x_i and x_j to the same value (i.e. $Pr(h(x_i) = h(x_j))$)? Lets call this lower bound p_1.

Similarly, if $d(x_i, x_j) \geq cr$ (for $c \geq 1$), what is an upper bound for the probability that h maps x_i and x_j to the same value (i.e. $Pr(h(x_i) = h(x_j))$)? Lets all this upper bound p_2.

Is there an inequality relationship between p_1 and p_2 and if so, what is it?

Problem 2(c). [2 points] LSH refers to the following properties. A family of hash functions is (r, c, p_1, p_2)-LSH ($1 \geq p_1 \geq p_2 > 0$, $c > 1$, $r > 0$) if:

1. $Pr(h(x_i) = h(x_j)) \geq p_1$ when $d(x_i, x_j) \leq r$;
2. $Pr(h(x_i) = h(x_j)) \leq p_2$ when $d(x_i, x_j) \geq cr$.

Note that h is a randomly sampled hash function from the family of hash functions and thus h is a random variable (even though it is a lower case, so formally, we might want to call it capital "H" to indicate that it

is a random variable, but we're using H as the family, and for consistency with existing literature, we use lower case for the random h). Intuitively, when two data points are close ($d(x_i, x_j) \leq r$), the hash function maps them to the same output value with (relatively) high probability at least equal to p_1. If two data points are far away (meaning $d(x_i, x_j) \geq cr$), then the hash function maps them to the same output value with (relatively) low probability at most equal to p_2. Also, note that $p_2 \leq p_1$, so that is the only relationship that designates that p_1 is "high" and p_2 is "low."

The very simple hash functions from H introduced above indeed satisfies the LSH property (you can verify by comparing your answers to the two previous questions). With the dataset $\mathbf{X}$, we can first hash all data points using a single function h randomly sampled from H. Then, given a query point q, we hash q with the same function h, and retrieve all data points in $\mathbf{X}$ that get hashed to the same value as $h(q)$ (if any). Finally, we can iterate over the retrieved points (if non-empty), and return the point closest to the query point q, thus achieving a computational savings. As $p_1 \geq p_2$, the hash function is more likely to hash close points into the same value than distant points.

The difference between p_1 and p_2, however, might not be appreciable. One simple trick to make it more probable that close (rather than distant) points map to the same value is to sample multiple hash functions from the family H; then, two data points have the same hashed value if all the sampled hash functions give the same value. In other words, we define a new hash function g via the use of k independent and uniformly random sampled hash functions from H, and we do this by concatenating their output together into a vector of length k.

$$g(x_i) = (h_1(x_i), h_2(x_i), h_3(x_i), \ldots, h_k(x_i)). \tag{6}$$

Give a lower bound for the probability that $g(x_i) = g(x_j)$ if $d(x_i, x_j) \leq r$ in terms of p_1 and k. Give an upper bound for the probability that $g(x_i) = g(x_j)$ if $d(x_i, x_j) \geq cr$ in terms of p_2 and k.

Note that $g(x_i) = g(x_j)$ if they are equal for every coordinate of the output vectors. You can also think the binary vector output of g as a binary encoding of an integer, so the output of g becomes an integer value.

Problem 2(d). [2 points]

As we increase the value of k, the close data points become more probable relative to the distant data points. However, we also get lower probability to have two data points hashed to the same value. For large value of k, the probability can be negligible, no two points may get hashed to the same value, and as a result, our algorithm may always return nothing. To alleviate such problems, we can have l instances of a g function, where each g function has been formed via k independently sampled hash functions from H. Then we hash the query point q with each of the l different g functions, and collect all data points (if any) that share the same hashed value as $g(q)$. Finally, we iterate over the collected data points from all g functions, and return the one with the least distance.

Give a lower bound for the probability that $\exists b \in \{0, 1, \ldots, l-1\}$ such that $g_b(x_i) = g_b(x_j)$ whenever $d(x_i, x_j) \leq r$, where g_b is the b^{th} such function.

Give an upper bound for the probability that $\exists b \in \{0, 1, \ldots, l-1\}$ such that $g_b(x_i) = g_b(x_j)$ if $d(x_i, x_j) \geq cr$.

Problem 2(e). [7 points] Again, assume we set r appropriately so that there exists some data point $x \in \mathbf{X}$ with $d(x, q) \leq r$.

Let $\rho = \frac{ln(p_1)}{ln(p_2)}$, $l = n^\rho$ and $k = \frac{ln(n)}{ln(1/p_2)}$. Consider the following two events:

1. For some $x' \in \mathbf{X}$, $d(x', q) \leq r$, $\exists b \in \{0, 1, \ldots, l-1\}$ such that $g_b(x') = g_b(q)$.
2. There are at most $4l$ items in $\mathbf{X}$ where each x in those $4l$ items has $d(x, q) \geq cr$ and for some b, $g_b(x) = g_b(q)$.

Show the first event (lets call it event (i)) happens with probability at least $1 - e^{-1}$ (note that $\lim_{k\to\infty}(1 - 1/k)^k = e^{-1}$ and $(1 - 1/k)^k < e^{-1}$). Show the second event (called event (ii)) happens with probability

at least $3/4$ (HINT: use Markov's inequality https://en.wikipedia.org/wiki/Markov%27s_inequality). Give a lower bound on the probability that both events happen.

Problem 2(f). [2 points] The two events from the previous question happen with a constant probability. We can then boost the probability of success by running multiple independent instances so that the probability that the two events fail for all instances is negligible. For this problem, assume that the previous two events happen with certainty.

Recall that we construct l hash functions, and each hash function g consists of k sampled functions from H. Given a query q, we collect all data points in $\mathbf{X}$ if they share the same hashed value for any g_b with the query point. Now we want to iterate over the collected points and report one point as the nearest neighbor. If we only want to return a point that has distance at most cr to the query point q, how many points do we need to check? Are we guaranteed that there is a point with distance at most cr and why?

Problem 3. Programming: Random Forests [20 points]

Problems 3 and 4 are really two parts of one problem, so you will definitely need to read the description of both problems 3 and 4 in tandem. We suggest reading both problems 3 and 4 several times before proceeding to work on either of them since there are various forward and backward references between the problems in the description below. That is, on first reading, a few terms will be encountered that are not fully defined until problem 4.

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 (RFs do not use a boosting like procedure, like GBDTs do that we describe below). 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 based on a a single feature dimension (thresholding in particular). We do this multiple times. When we're all done, RFs produce predictions by averaging the outputs of all the B models as follows:

$$\hat{y}_i = \frac{1}{B}\sum_{b=1}^{B} f_b(x_i;\theta_b). \tag{7}$$

The optimization problem for training tree b in a RF is

$$\min_{\theta_b} \sum_{i=1}^{n} \ell(y_i,\hat{y}_i) + \Omega(\theta_b), \tag{8}$$

where $\hat{y}_i = f_b(x_i;\theta_b)$ is the prediction produced by tree-b on data point x_i; $\ell(\cdot,\cdot)$ is a loss function (described in Section 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-b). Most descriptions of ensemble learning in Section 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. (9) and Eq. (10).

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 4) is the tree growing methods. In the case of random forests, we greedily learn each tree using a bootstrapped data sample and random feature selection as described in class (please find the lecture slide where this was done), while in the case of GBDTs, we use the standard greedy tree-splitting algorithm (also described in class) but in a different way. That is, the key difference is the data that is being used (the original data in the case of GBDT, or bootstrap samples for each tree in the case of RFs). 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, in order to facilitate code reuse between this and the next problem, and also to make a more fair comparison between RFs and GBDTs, we ask you to use the same code base between this and the next problem (detailed in Section4 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 starts 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 RD only depend on $\hat{y}_i^0$, not $\hat{y}_i^{k-1}$. Instructions 2-5 in Section 5, however, can be still applied to RF tree building here.

In this problem, you will implement RFs for both regression and binary classification problems. Please read Section 1, 4 and 5 below before you start.

Problem 3(a). [10 points] Implement the RF procedure for the 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 comparing to least square regression and ridge regression? Include a few comparison plots to support what you find.

Problem 3(b). [10 points] Implement the RF procedure for the binary classification task, and test its performance on Credit-g dataset. This is a dataset for classifying people as either a good or bad credit risk, based on 20 attributes. The full description of the attributes can be found at https://www.openml.org/d/31. Report both training and test accuracy. Also, try your implementation on the breast cancer diagnostic dataset[2], and report both training and test accuracy.

Problem 4. Programming: Gradient Boosting Decision Trees [30 points]

Problems 3 and 4 are really two parts of one problem, so you will definitely need to read the description of both problems 3 and 4 in tandem. As mentioned above (and again repeated here), we suggest reading both problems 3 and 4 several times before proceeding.

1 Problem of Ensemble Learning in GBDTs

Gradient Boosting Decision Trees (GBDT) are 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:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i;\theta_k). \tag{9}$$

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.

GBDTs have been widely used in a variety of real applications and industrial fields with great success. This is due to GBDT's training efficiency (especially compared to deep neural networks), and more importantly, its interpretability, since decision trees and the decision rules embedded within them are much easier for humans to understand and explain than the high-dimensional model weights and artificial non-linearly aggregated features produced by complicated deep neural networks (see for example https://microscope.openai.com/models/inceptionv1/mixed5b_5x5_0/34).

In addition, GBDT methods have been the winner of a good number of machine learning competitions, for example, Kaggle competitions[3], where people can earn prizes by winning an ML competition and some

[1] https://www.kaggle.com/c/boston-housing

[2] https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

[3] https://www.kaggle.com/competitions

do simply by using GBTs (via standard open-source packages such as XGBoost or LightGBM). Furthermore, GBDTs (more so than many other methods) have empirically been shown to be robust to noise and data imbalance issues on various tasks. There are many fast GBDT packages (i.e., with optimized efficiency for each step of GBDT building and using) and this makes using GBDTs easy and fast to use — for example, see XGBoost[4] and LightGBM[5]. As a result, GBDTs are a useful state-of-the-art ML tool. As mentioned in class, given the success of deep neural networks, it is heartening that there is a class of methods that is different but that often performs so well. In the below, you will learn to implement GBTDs step by step, by following the tutorial below.

Important: you are **not allowed** to use any GBDT package, or any pre-existing GBDT software or code, in your code solutions — that is, you need to implement GBDTs from scratch (and we will check that you have done so).

The optimization problem for training an ensemble of models is

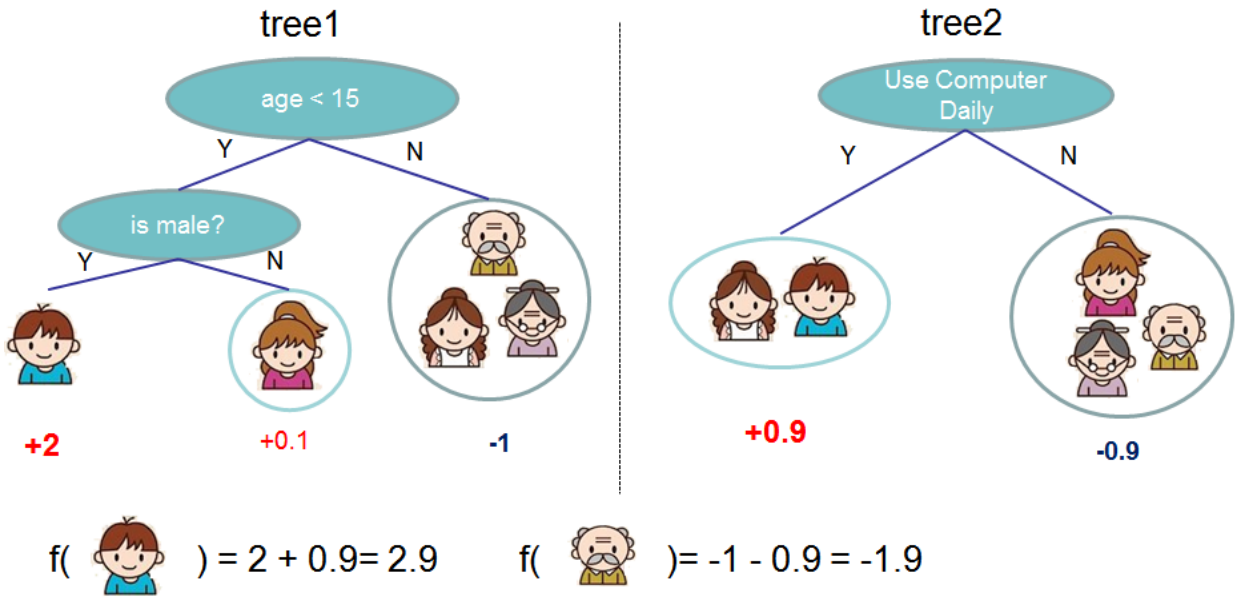
$$\min_{\{\theta_k\}_{k=1}^K} \sum_{i=1}^n \ell(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(\theta_k), \tag{10}$$

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 the overall ensemble loss function (detailed definitions are given in Section 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).

With GBDTs, 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 $|L_k|$-dimensional vector storing the weights of all the leaf nodes. We may thus, deceptively simply, write the decision tree function as follows

$$f_k(x) = w^k_{q_k(x)}, \tag{11}$$

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.



[4] https://github.com/dmlc/xgboost
[5] https://github.com/Microsoft/LightGBM

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

Therefore, to define a tree $f_k(\cdot)$, we need to determine: (1) the structure of the tree $\mathcal{T} \triangleq (N_k \cup L_k, E)$ where E is the set of tree edges; (2) the feature dimension p_j and threshold τ_j associated with each non-leaf node $j \in N_k$; (3) 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.,

$$\theta_k = \left\{ \mathcal{T}, w^k, \{(p_j, \tau_j)\}_{j \in N_k} \right\}. \tag{12}$$

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 standard naive gradient descent is inapplicable.

2 Overview of the GBDT algorithm

To complement the description below, you might find it useful to take a look at the following to links to visualizations of the GBDT method.
http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html and http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html
At these sites, you can choose different toy datasets and tune GBDT parameters to see how the learnt trees can fit the data.

The basic idea of GBDT training is additive training, or boosting. As mentioned above, and described in class, Boosting is a meta-algorithm that trains multiple models one after another, and in the end combines them 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] (as we went over in class), and gradient boosting [3, 4], as well as others. This is distinct from random forests [5, 2] described above. 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. Also, see lecture class slides.

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 a coordinate descent algorithm (an example of which you implemented for Lasso in Homework 3), each outer loop iteration requires one pass over all feature dimensions (or a random subset thereof). In each inner loop iteration, it starts from the first dimension, and optimizes only one dimension of the weight vector by fixing (and conditioning on) all the other dimensions. Each tree in a 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 at a time time by fixing, and conditioning 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, GBDT would optimize each coordinate only once in succession in a single pass over each coordinate.

Each time we optimize a tree, conditioned on all previously produced trees, we use the greedy strategy as we spoke about in class. After training of a given 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 firstly 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 also be empty, and so this also explains how to start with the first tree).

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

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

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

That is, each tree's prediction is based on fixing up the (additive) aggregation of all the previous trees predictions. Conditioning on the predictions of the previous $k-1$ trees, the objective used to optimize tree-k is the following:

$$F_k(\theta_k) = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i^k) + \Omega(\theta_k) = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i^{k-1} + f_k(x_i)) + \Omega(\theta_k), \tag{13}$$

Note that this consists of a sum, over the training set, of the loss plus a regularizer, but the loss is based on how much better the k^{th} tree can do starting from the aggregation of the previous trees $\hat{y}_i^{k-1}$, and the regularizer applies to the k^{th} tree.

Let's simplify the first term using Taylor's expansion, ignoring higher order terms (which is, after all, how we always use Taylor's series):

$$f(x+\Delta) \approx f(x) + f^{'}(x)\Delta + \frac{1}{2} f^{''}\Delta^2. \tag{14}$$

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

$$\ell(y_i, \hat{y}_i^{k-1} + f_k(x_i)) \approx \ell(y_i, \hat{y}_i^{k-1}) + g_i f_k(x_i) + \frac{1}{2} h_i f_k^2(x_i), \tag{15}$$

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.,

$$g_i \triangleq \frac{\partial \ell(y_i, \hat{y}_i^{k-1})}{\partial \hat{y}_i^{k-1}}, \quad h_i \triangleq \frac{\partial^2 \ell(y_i, \hat{y}_i^{k-1})}{(\partial \hat{y}_i^{k-1})^2} = \frac{\partial g_i}{\partial \hat{y}_i^{k-1}}. \tag{16}$$

The second term $\Omega(\theta_k)$ in Eq. (13) is a regularization term aiming to penalize the degree of complexity of tree-k. It is based on the number of leaf nodes (which in general we want to encourage to be small), and

the L2 regularization of the weights w^k (which we also want to keep small). With GBDTs, it is defined as follows:

$$\Omega(\theta_k) \triangleq \gamma|L_k| + \frac{1}{2}\lambda\|w^k\|_2^2. \tag{17}$$

We plugin Eq. (15) and Eq. (17) into Eq. (13), and after ignoring constants, we get

$$\begin{aligned} F_k(\theta_k) + \text{const.} &\approx \sum_{i=1}^{n}\left[g_i f_k(x_i) + \frac{1}{2}h_i \cdot f_k^2(x_i)\right] + \gamma|L_k| + \frac{1}{2}\lambda\|w^k\|_2^2 \\ &= \sum_{i=1}^{n}\left[g_i w^k_{q_k(x_i)} + \frac{1}{2}h_i \cdot (w^k_{q_k(x_i)})^2\right] + \gamma|L_k| + \frac{1}{2}\lambda\|w^k\|_2^2 \\ &= \sum_{j\in L_k}\left[\left(\sum_{i\in I_j} g_i\right) w_j^k + \frac{1}{2}\left(\lambda + \sum_{i\in I_j} h_i\right)(w_j^k)^2\right] + \gamma|L_k|, \end{aligned} \tag{18}$$

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. (18) gives us the objective of optimizing a tree conditioned on all the previously achieved trees.

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. (18): 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

$$w_j^k = -\frac{G_j}{H_j + \lambda}, \text{ where } G_j \triangleq \sum_{i\in I_j} g_i, \text{ and } H_j \triangleq \sum_{i\in I_j} h_i. \tag{19}$$

We can plug the above optimal w_j^k into Eq. (18) and obtain an updated objective

$$F_k\left(\theta_k; \left\{w_j^k = -\frac{G_j}{H_j+\lambda}\right\}_{j\in L_k}\right) = \gamma|L_k| - \frac{1}{2}\sum_{j\in L_k}\frac{G_j^2}{H_j+\lambda} \tag{20}$$

Note that the left hand side of this equation simply means that it is $F_k(\theta_k)$ but where, in the expression for $F_k(\theta_k)$, w_j^k for all j and for each k is assigned to the appropriate (and given) values.

There are still, however, two groups of unknown parameters within θ_k — this includes the tree structure $\mathcal{T}$ itself, and also the decision rules $\{(p_j, \tau_j)\}_{j\in N_k}$. In the following, we will describe 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. (20), 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. It should be noted that this is similar to the training of DTs that we described in class where we used MSE gain (for regression) or information gain (for classification) to decide a coordinate and threshold to split each tree note. Here, the only real difference is that we're using the above conditional objective which is based on gradients; and since it is conditional, it is a form of boosting. Hence the name, gradient boosting.

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. (20). Before splitting node j to a left child $j(L)$ and a right child $j(R)$, the objective is

$$F_k\left(\theta_k; \left\{w_j^k\right\}_{j\in L_k}\right) = \gamma|L_k| - \frac{1}{2}\sum_{j'\in L_k\setminus j}\frac{G_{j'}^2}{H_{j'}+\lambda} - \frac{1}{2}\frac{G_j^2}{H_j+\lambda} \tag{21}$$

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

$$F_k\left(\theta_k; w_{j(L)}^k, w_{j(R)}^k, \left\{w_{j'}^k\right\}_{j'\in L_k\setminus j}\right) = \gamma(|L_k|+1) - \frac{1}{2}\sum_{j'\in L_k\setminus j}\frac{G_{j'}^2}{H_{j'}+\lambda} - \frac{1}{2}\frac{G_{j(L)}^2}{H_{j(L)}+\lambda} - \frac{1}{2}\frac{G_{j(R)}^2}{H_{j(R)}+\lambda} \tag{22}$$

Hence, the improvement (we also call it the "gain" here as well) is

$$F_k\left(\theta_k; \left\{w_j^k\right\}_{j\in L_k}\right) - F_k\left(\theta_k; w_{j(L)}^k, w_{j(R)}^k, \left\{w_{j'}^k\right\}_{j'\in L_k\setminus j}\right) \tag{23}$$

$$= \frac{1}{2}\left[\frac{G_{j(L)}^2}{H_{j(L)}+\lambda} + \frac{G_{j(R)}^2}{H_{j(R)}+\lambda} - \frac{G_j^2}{H_j+\lambda}\right] - \gamma \tag{24}$$

$$= \frac{1}{2}\left[\frac{G_{j(L)}^2}{H_{j(L)}+\lambda} + \frac{G_{j(R)}^2}{H_{j(R)}+\lambda} - \frac{(G_{j(L)}+G_{j(R)})^2}{H_{j(L)}+H_{j(R)}+\lambda}\right] - \gamma. \tag{25}$$

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. (20). That is, we wish to perform the following optimization:

$$\max_{(p_j,\tau_j)} \frac{1}{2}\left[\frac{G_{j(L)}^2}{H_{j(L)}+\lambda} + \frac{G_{j(R)}^2}{H_{j(R)}+\lambda} - \frac{(G_{j(L)}+G_{j(R)})^2}{H_{j(L)}+H_{j(R)}+\lambda}\right] - \gamma. \tag{26}$$

We starts from a 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. (26)) 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}.

5 Practical Implementation Details

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 in order to give more chance to new trees, we will make the process less greedy. In particular, we usually assign a weight $0 \leq \eta \leq 1$ to each newly added tree

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

when aggregating its output with the outputs of previously added trees. Hence, the sequence in the beginning of Section 3 becomes

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

Note this change needs to be applied in both training and also during testing/inference. $0 \leq \eta \leq 1$ is usually called the "learning rate" of GBDT, but it is not exactly the same as the variable we usually call 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 similar rule 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 training set. For classification, it is fine to also use this initialization (which is 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.,

$$\ell(y,\hat{y}) = y\log(1+\exp(-\hat{y})) + (1-y)\log(1+\exp(\hat{y})). \tag{27}$$

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

$$\Pr(class=1) = \frac{1}{1+\exp(-\hat{y})}, \quad \Pr(class=0) = 1 - \Pr(class=1). \tag{28}$$

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

$$class = \begin{cases} 1, & \Pr(class=1) > 0.5\\ 0, & \Pr(class=1) \leq 0.5 \end{cases} \tag{29}$$

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} requires 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 in the end of Section 4 fulfills.

6. **Acceleration:** We encourage you to apply different methods of acceleration after you making 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.**

6 Questions

Problem 4(a). [2 points] What is the computational complexity of optimizing a tree of depth d in terms of m and n?

Problem 4(b). [2 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.

Problem 4(c). [2 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")

Problem 4(d). [2 points] What are the major differences of GBDT, comparing to gradient descent, stochastic gradient descent, and coordinate gradient descent algorithms, in the way of updating model parameters? Please list at least two differences.

Problem 4(e). [10 points] Implement GBDT for 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 comparing to least square regression and ridge regression?

Problem 4(f). [10 points] Implement GBDT for binary classification task, and test its performance on Credit-g dataset. Report the training and test accuracy. Try your implementation on breast cancer diagnostic dataset, and report the training and test accuracy.

Problem 4(g). [2 points] According to the results on the three experiments, how is the performance of random forests comparing 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.