Homework 4: Decision Trees and Ensemble Methods

Due: Tuesday, April 14th, 2026 at 11:59pm ET

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX or MathJax in iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. For the coding problems, the text **Submission:** indicates all you need to submit in your PDF submission.

Note: Compared to the previous homeworks, this is a relatively short homework (two main Problems) that is composed mostly of coding problems. All the coding problems can be done using the code from `ps4_skeleton.ipynb`. One new package you may have to install for this assignment is `graphviz`, which is just used to visualize the decision trees.

Problem 1: Decision Trees (50 points)

In this problem we'll implement decision trees from scratch for both classification and regression. The strategy will be to implement a generic class, called `Decision_Tree`, which we'll supply with the impurity function we want to use to make node splitting decisions, as well as the estimator we'll use to come up with the prediction associated with each leaf node. For classification, this prediction could be a vector of probabilities, but for simplicity we'll just consider hard classifications here.

Recall from lecture that we can use decision trees to classify data by recursively splitting the data into two subsets based on a feature and a threshold. The impurity function we use to make the split decision is the entropy or the Gini index.

To handle multiclass classification, where $\mathcal{Y} = \{1, 2, \ldots, K\}$, we can use the entropy or the Gini index to make the split decision. Using the notation from class, let node m represent region (leaf node) R_m and let p_{mk} be the fraction of training examples in R_m that belong to class k. Then the entropy of region R_m is given by:

$$H(R_m) = -\sum_{k=1}^{K} p_{mk} \log_2(p_{mk})$$

and the Gini index of region R_m is given by:

$$G(R_m) = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}).$$

When $K = 2$ (as in the code for this problem), the entropy and Gini index are given by:

$$H(R_m) = -p_{m1} \log_2(p_{m1}) - p_{m2} \log_2(p_{m2})$$

$$G(R_m) = p_{m1}(1 - p_{m1}) + p_{m2}(1 - p_{m2}).$$

Problem 1(a) (5 points)

Write a function called `compute_entropy` that takes as input a numpy array of binary labels and returns the entropy of the labels. Your submitted code should work for both binary and multiclass classification (although we will focus on the binary case for this problem).
Submission: Submit the code for `compute_entropy`.

Problem 1(b) (5 points)

Write a function called `compute_gini` that takes as input a numpy array of binary labels and returns the Gini index of the labels. Your submitted code should work for both binary and multiclass classification (although we will focus on the binary case for this problem).
Submission: Submit the code for `compute_gini`.

We will now complete the class `Decision_Tree`, given in the skeleton code. The high-level intended implementation is as follows: Each object of type `Decision_Tree` represents a single node of the tree. The depth of that node is represented by the variable `self.depth`, with the root node having depth 0.

The main job of the `fit` function is to decide, given the data provided, how to split the node or whether it should remain a leaf node. If the node will split, then the splitting feature and splitting value are recorded, and the left and right subtrees are fit on the relevant portions of the data. Thus tree-building is a recursive procedure. We should have as many `Decision_Tree` objects as there are nodes in the tree. We will not implement pruning here. Some additional details are given in the skeleton code.

Each `Decision_Tree` object has the following parameters:

- `split_loss_function`: This refers to the method for splitting the node (the impurity score for classification; the MSE or MAE for regression).
- `leaf_value_estimator`: This refers to the method for estimating the value of that each leaf node predicts (the most common label for classification; the mean/median for regression).
- `depth`: This refers to the depth of the current node (recall each `Decision_Tree` object represents a single node of the tree). The default value should be 0, which represents the root node of the tree.

- `min_sample`: This refers to a global parameter for the entire decision tree that should be passed from node to node. It is the minimum number of samples we allow in a node before we stop splitting. This is one of the two stopping criterion for the tree building process. The other stopping criterion is the maximum depth of the tree, which is represented by the `max_depth` parameter.

- `max_depth`: This refers to a global parameter for the entire decision tree that should be passed from node to node. It is the maximum depth of the tree. This is one of the two stopping criterion for the tree building process. The other stopping criterion is the minimum number of samples required to split a node, which is represented by the `min_sample` parameter.

- `is_leaf`: This refers whether the current node is a leaf or not; it is Boolean valued.

- `value`: This parameter should only be set to something other than `None` if the node is a leaf node. This refers to the actual value $\hat{y}$ that the leaf node predicts for all examples that fall into the leaf.

- `split_id`: This parameter refers to the feature ID $j \in \{0, \ldots, d-1\}$ that the node splits on. If the node is not a leaf, this should be an integer in $\{0, \ldots, d-1\}$; if it is a leaf, it should take the default of -1.

- `split_value`: This parameter refers to the feature value that the node splits on. If the node is not a leaf, this should be a float; if it is a leaf, it should take the default value of `np.inf`.

- `left`: This parameter refers to the `Decision_Tree` object corresponding to the left subtree. This subtree should be fit on the data that fall to the left of `self.split_value`.

- `right`: This parameter refers to the `Decision_Tree` object corresponding to the right subtree. This subtree should be fit on the data that fall to the right of `self.split_value`.

The `Decision_Tree` class has the following methods:

- `fit`: This method is used to fit the decision tree to the data. It is a recursive procedure that builds the tree. It will call the `find_best_feature_split` method to find the best feature to split on and the best split value for a node.

- `find_best_split`: For a fixed feature, this method is used to find the best split value for a node.

- `find_best_feature_split`: For a given node, this method is used to find the best feature to split on and the best split value and it will call the `find_best_split` method to find the best split value for the feature.

We will first implement the `find_best_split` method. In words, this method takes in a feature $j \in \{1, 2, \ldots, d\}$ and returns the best split value for that feature for the data in the node: `X_node` and `y_node`. For example, if $d = 2$, then the feature index $j = 1$ corresponds to the first column of `X_node`, and the feature index $j = 2$ corresponds to the second column of `X_node`. The function should take in either $j = 1$ or $j = 2$ as input as the `feature_id` parameter and iterate over all the unique values of the feature to find the best split value, which should occur at midpoints between unique values. Because Python is zero-indexed, your code should actually take in $j \in \{0, \ldots, d-1\}$ as input.

Note: You may assume in your implementation that the features are continuous real values. This will make it easier to determine the best split value per feature.

Problem 1(c) (10 points)

Complete the `find_best_split` method. For a fixed feature, this method is used to find the best split value for a node. It takes in the data in the node: `X_node` and `y_node`, and the feature index `feature_id`, which is simply the (integer) index $j \in \{0, \ldots, d-1\}$ of the feature to split on.
The function should return the best split value `best_split_value` and the corresponding loss/impurity of the split `best_loss`, which are both floats.
Hint: Referring to the *Splitting Criterion & Overfitting* slides from the lecture notes (pg. 11-21) should help with implementing this method. Paticularly, focus on slides 14 and 21.
Submission: Submit the code for `find_best_split`.

With the `find_best_split` method implemented, we can now implement the `find_best_feature_split` method. In words, this method takes in the data in the node: `X_node` and `y_node` and returns the best feature to split on *and* the best split value for that feature. The function should iterate over all the features and call the `find_best_split` method to find the best split value for each feature. It should return the best feature index `best_feature_id` and the best split value `best_split_value`.

Problem 1(d) (5 points)

Complete the `find_best_feature_split` method. For a given node, this method is used to find the best feature to split on and the best split value and it will call the `find_best_split` method to find the best split value for the feature.
It takes as input the data in the node: `X_node` and `y_node`. It should output the best feature index `best_feature_id` (which should be an integer between 0 and $d-1$) and the best split value `best_split_value` (which should be a float).
Hint: This function should be relatively straightforward after implementing the `find_best_split` method.
Submission: Submit the code for `find_best_feature_split`.

We will now complete the class `Decision_Tree` by implementing the `fit` method, which works *recursively*. There are two stopping criteria for the tree building process: the minimum number of samples required to split a node, which is represented by the `min_sample` parameter, and the maximum depth of the tree, which is represented by the `max_depth` parameter. If neither of these are met yet, the `fit` method should call `find_best_feature_split` to find the best feature to split on and the best split value for the node.

If either stopping criterion is met (or we cannot find a feature to split on that decreases loss/impurity), we designate the current `Decision_Tree` object as a leaf node by setting `self.is_leaf = True`. In this case, we can use `self.leaf_value_estimator` to compute `self.value`, the predicted $\hat{y}$ for any examples falling into this leaf.

If neither stopping criterion is met and we split, then we should take the returned `best_feature_id` and `best_split_value` from `find_best_feature_split` and assign them to `self.split_id` and `self.split_value`. Then, we should make two subtrees, which are themselves `DecisionTree` objects, and initialize them appropriately with the data corresponding to the split. Finally, we should continue the recursion by calling `.fit` on both subtrees.

Problem 1(e) (10 points)

Complete the class `Decision_Tree` by implementing the `fit` method. This should implement the recursive procedure described above. It should take as input `X` and `y`, the original data. It should return `self`, a `DecisionTree` object.
At the end of calling `fit`, the `DecisionTree` object referred to from `self` should be the root node of a constructed Decision Tree.
Submission: Submit the code for `fit`.

Problem 1(f) (10 points)

Run the code provided that builds trees for the two-dimensional classification data; the parameters you should use are: `min_sample = 5` and `max_depth` is up from 1 to 6. The code should train trees with depths 1, 2, 3, 4, 5, and 6. For each depth, the code plots the decision regions of the tree. There should be 6 plots total, one for each depth. Call these Plots 1, 2, 3, 4, 5, and 6 and submit these.

The behavior should be qualitatively the same to the `sklearn` implementation of `DecisionTreeClassifier`; we have included code to compare the decision boundaries of the trees we build with the `sklearn` implementation. Double-check this before submitting.

Discuss the effect of increasing the depth of the tree on the decision boundaries. What do you observe? What might constitute *underfitting* and *overfitting* in this context? Provide a 3-4 sentence written answer to these questions.

You can also visualize the fit decision trees using the provided code; no need to turn in the visualization. For visualization, you may need to install `graphviz`.

Submission: Submit only Plots 1-6 (no code) for the decision boundaries and your written discussion.

Problem 1(g) (5 points)

If you completed Problem 1(f), correctly, you may have observed that none of the trees we built seem to (qualitatively) fit the data well. You may have seen this, intuitively, by looking at the decision boundaries and noticing that they are not a strict rectangle in the center of the plots and possibly overfitting to noisy points (or underfitting the data at the center).

Experiment with different values of `min_sample` and `max_depth` to see if you can get the trees to fit the data better (qualtitatively, this should result in a rectangle that is strictly in the center of the plot). You may use the built-in `sklearn` implementation of `DecisionTreeClassifier` to help you experiment with different parameters (your implementation should be qualitatively the same to the `sklearn` implementation).

Submit a plot of a decision boundary you achieved that fits the data better with your new parameters and a 3-4 sentence written answer to the question of what parameters you used and why you chose them.

Submission: Submit your plot and your written discussion.

Problem 2: Ensembling via Gradient Boosting (50 points)

Recall the general gradient boosting algorithm, for a given loss function ℓ and a hypothesis space $\mathcal{H}$ of regression functions (i.e. functions mapping from the input space to $\mathbb{R}$):

0: Initialize $f_0(x) = 0$.

1: For $t = 1$ to T:

(a) Compute:

$$\mathbf{g}_t = \left(\frac{\partial}{\partial f_{t-1}(x^{(j)})} \sum_{i=1}^{n} \ell\big(f_{t-1}(x^{(i)}), y^{(i)}\big) \right)_{j=1}^{n}$$

where $\mathbf{g}_t \in \mathbb{R}^n$ is the gradient vector of the objective with respect to $f_{t-1}(x^{(j)})$. In this problem, this vector is known as the vector of *pseudo-residuals*.

(b) Fit regression model to $-\mathbf{g}_t$:

$$h_t = \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n} \big((-\mathbf{g}_t)_i - h(x^{(i)})\big)^2 .$$

(c) Choose fixed step size $\nu_t = \nu \in (0, 1]$, or take

$$\nu_t = \arg\min_{\nu > 0} \sum_{i=1}^{n} \ell\left(f_{t-1}(x^{(i)}) + \nu h_t(x^{(i)}), y^{(i)}\right)$$

(d) Take the step:

$$f_t(x) = f_{t-1}(x) + \nu_t h_t(x)$$

2: Return f_T.

This method goes by many names, including gradient boosting machines (GBM), generalized boosting models (GBM), AnyBoost, and gradient boosted regression trees (GBRT) when $\mathcal{H}$ is some class of decision trees, among others. One of the nice aspects of gradient boosting is that it can be applied to any problem with a (sub)differentiable loss function (recall our discussion of subgradients in Problem Set 3).

We'll keep things simple and consider the standard regression setting with square loss. In this case, we have $\mathcal{Y} = \mathbb{R}$, our loss function is given by $\ell(\hat{y}, y) = 1/2\,(\hat{y} - y)^2$, and at the t'th round of gradient boosting, we have

$$h_t = \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n} \big(y^{(i)} - f_{t-1}(x^{(i)}) - h(x^{(i)})\big)^2 . \tag{1}$$

This happens to be the choice of h_t that is appropriate for the square loss in the step (b) above.

Problem 2(a) (5 points)

Using Steps 1(a) and 1(b) of the general gradient boosting algorithm, derive the equation for h_t in the case of square loss shown above.
Hint: To derive $\mathbf{g}_t$, treat each $f_{t-1}(x^{(i)})$ as a constant and try to derive it entry-by-entry (there are n entries in the gradient vector).

Notice by this form of Equation (1) that we are essentially fitting a regression tree to the residuals of the previous model. We will now make sure the skeleton code provided from Problem 1 implements the `RegressionTree` functionality.

Problem 2(b) (5 points)

Using the provided skeleton code, verify that the `Decision_Tree` class you implemented in Problem 1 also works for regression. There is nothing to code here; just run the cell that trains decision trees of `max_depth` from 1 to 6 and attach the plots generated (there should be six in one figure for depths 1 to 6) to this submission and attach the plots.
You can double-check correctness with the code provided that your results qualitatively match the `sklearn` implementation of `DecisionTreeRegressor` when the `criterion` is `squared_error`.
Submission: Just the six plots, where `max_depth` varies from 1 to 6.

We will now move on to the main part of the problem, implementing the gradient boosting algorithm, which is a special case of the general gradient boosting algorithm where the base regression algorithm is a regression tree (using our implementation `RegressionTree` built atop Problem 1).

We will incrementally fill out the `gradient_boosting` class, whose parameters are:

- `n_estimator`: number of estimators (i.e. number of rounds of gradient boosting T).
- `pseudo_residual_func`: function used for computing pseudo-residual ($\mathbf{g}_t$) between training labels and predicted labels at each iteration.
- `learning_rate`: step size of gradient boosting, kept constant for all iterations (i.e. $\nu_t = \nu$).
- `min_sample`: minimum number of samples required to split an internal node for the base model (assumed to be RegressionTree).
- `max_depth`: maximum depth of the base model (assumed to be RegressionTree).
- `estimators`: list of base models (RegressionTree). These are $h_1, h_2, \ldots, h_T$.

Problem 2(c) (5 points)

Implement the `pseudo_residual_L2` function to compute the pseudo-residuals $\mathbf{g}_t$ for the square loss.
Hint: This should be very straightforward (one line of code); don't overthink it!
Submission: Submit the code for `pseudo_residual_L2`.

We can now implement the `fit` method. Notice that, at the core of the fit method, we are fitting a regression tree to the pseudo-residuals $\mathbf{g}_t$. This is the same as the `fit` method for the `RegressionTree` class, but we are using the pseudo-residuals instead of the true labels.

Problem 2(d) (5 points)

Show that Equation (1) is equivalent to an ERM problem:

$$h_t = \arg\min_{h \in \mathcal{H}} \sum_{i=1}^{n} \ell(h(x^{(i)}), \tilde{y}^{(i)}),$$

where you should specify what $\tilde{y}^{(i)}$, the hypothesis class $\mathcal{H}$, and the loss function ℓ are.

Although we know from class that a decision tree doesn't necessarily minimize the squared loss as an ERM problem (it uses a top-down greedy algorithm to minimize errors), we can still use `RegressionTree` to fit the base model at h_t as an approximation to the argmin in Equation (1). To do, this we simply use what we have implemented already and just call the `fit` method of `RegressionTree` with the pseudo-residuals as the labels.

Problem 2(e) (10 points)

Implement the `fit` method for the `gradient_boosting` class. This should implement the algorithm described above, with a fixed step size $\nu_t = \nu$.
It should take as input `X` and `y`, corresponding to training data $X \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^{n \times 1}$. The variables `X` and `y` are numpy arrays of shape (n, d) and (n, 1), respectively. It should return `self`, a `gradient_boosting` object.
Submission: Submit the code for `fit`.

Problem 2(f) (10 points)

Implement the `predict` method for the `gradient_boosting` class. This should return the predicted values for any given data $X \in \mathbb{R}^{n \times d}$, where X is represented as `X` in the code, a numpy array of shape (n, d). It should return a numpy array of shape (n, 1) containing the predicted values.
Submission: Submit the code for `predict`.

Problem 2(g) (5 points)

Use the provided code to create six plots of gradient boosting models on the regression dataset for values of `n_estimator` in $\{1, 5, 10, 20, 50, 100\}$.
Attach the plots to this submission and discuss the effect of increasing the number of estimators (T) on the generated model. Provide a 2-3 sentence written answer to these questions.
Submission: Submit the six plots and your written discussion.

Problem 2(h) (5 points)

This problem is relatively open-ended. Write your own code to experiment with the parameter `n_estimator` and how it affects the training error (on `regression-train.txt`) vs. test error (on `regression-test.txt`), in terms of squared loss. Do this for at least a range of values of `n_estimator` in $\{1, 5, 10, 20, 50, 100\}$.
What is a good value of `n_estimator`? What is a bad value of `n_estimator`? Explain your reasoning.
Submission: Submit a written answer to this question along with a plot of the training error vs. test error for the different values of `n_estimator`.