CS5228: Assignment

1. Introduction

In this assignment, you are going to implement a Decision Tree Regressor, a Gradient Boosting Regressor, and a Hierarchical Clustering Algorithm. The deadline for this assignment is **5pm on 21/Oct/2019**.

2. Decision Tree Regressor

In this part, you are going to implement a Least Squares Regressor Tree. We have provided the data and the test cases to you. You should use **Python 3.X** in your implementation.

CART (Classification and Regression Trees), a non-parametric statistical algorithm, is developed by Breiman, Friedman, Olshen, Stone in early 80's. CART can be used to predict or analyze both categorical (classification) and continuous or numerical (regression) data. The process of growing the Least Squares Regressor Tree by CART is summarized as follows:

Input: Training Dataset D;

Output: A regression decision tree $f(x)$.

In the input space of the training dataset, each region is recursively divided into two sub-regions and the output values on each sub-region are determined to construct a binary decision tree.

Steps:

1. Select the optimal splitting variable j and the splitting threshold s to solve

$$\min_{j,s}\left[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{x_i\in R_2(j,s)}(y_i-c_2)^2\right]$$

   Traverse the splitting variable j and scan the splitting threshold s for the fixed splitting variable j, and determine the pair (j,s) that minimizes the expression. If we collect the values of variable j from all training sample and denote it as V_j, then the splitting threshold s is one element of V_j. For this assignment, if you find that every value in $[a_1,\ a_2)$ (where a_1 and a_2 are from V_j) can be a splitting threshold with the same minimum impurity, then a_1 should be selected as the threshold. You should not choose other thresholds, e.g., $(a_1+a_2)/2$. This is requirement is for automatic grading.

2. Split the region with the selected pair (j,s) and determine the corresponding output value:

$$\mathrm{R}_1(j,s)=\{x|x^{(j)}\le s\},\mathrm{R}_2(j,s)=\{x|x^{(j)}>s\}$$

$$\hat{c}_m=\frac{1}{N_m}\sum_{x_i\in R_m}y_i\,,\qquad x\in R_m,\qquad m=1,2$$

3. Repeat the above two steps considering each resulting region as a parent node until the maximum depth of the tree is obtained.

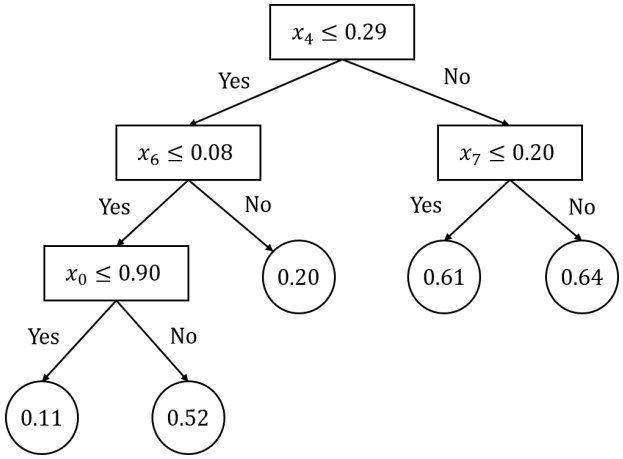
4. The input space is divided into M regions $\mathrm{R}_1,R_2,\dots,R_M$, then the decision tree is

$$f(x)=\sum_{m=1}^{M}\hat{c}_m I(x\in R_m)$$

Implementation details:

You are required to implement the methods "`fit(self, X, y)`" and "`predict(self, X)`" in the file "DecisionTreeRegressor.py". You are not allowed to modify the definition of the given attributes and methods, i.e. no name change and no additional parameters in the methods. But you can add other methods or attributes in the class for your convenience.

In the function "`fit(self, X, y)`", you should update the `self.node`, whose type is dictionary. There are four keys in `self.node`, which are "`splitting_variable`", "`splitting_threshold`", "`left`" and "`right`". The value of "`splitting_variable`" should be an integer number. The value of "`splitting_threshold`" should be a float number. The values of "`left`" and "`right`" should be either a float number or a dictionary.
For example,



the above decision tree can be represented by

```
self.root = {"splitting_variable": 4,
             "splitting_threshold": 0.29,
             "left": {"splitting_variable": 6,
                      "splitting_threshold": 0.08,
                      "left": {"splitting_variable": 0,
                               "splitting_threshold": 0.90,
                               "left": 0.11,
                               "right": 0.52},
                      "right":0.20}
             "right": {"splitting_variable": 7,
                       "splitting_threshold": 0.20,
                       "left": 0.61,
                       "right": 0.64}
             }
```

Noted that the floating point precision issue will be considered when grading. In other words, if your results are slightly different (e.g., less than 10^{-5}) to the correct answer due to the precision issue, do not worry.

3. Gradient Boosting Regressor

In this part, you are going to implement a Gradient Boosting Regressor based on the Least Squares Regressor Tree you have implemented in part 2. We have provided the data and the test cases to you. You should use **Python 3.X** in your implementation.

In this task, the loss function is defined as

$$L\big(y, f(x)\big) = \frac{1}{2}\big(y - f(x)\big)^2$$

Then, the negative gradient of $L\big(y, f(x)\big)$ is

$$-\frac{\partial L\big(y_i, f(x_i)\big)}{\partial f(x_i)} = y_i - f(x_i)$$

The algorithm of Gradient Boosting Algorithm is summarized as follows:

Input: Training dataset $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}, x_i \in \mathcal{X} \in \mathbf{R}^n, y_i \in \mathcal{Y} \in \mathbf{R}^n$; Learning rate lr**;** Loss function $L(y, f(x))$.

Output: A regression decision tree $\hat{f}(x)$.

Steps:

1. Initialize $f_0(x) = argmin_c \sum_{i=1}^{N} L(y_i, c)$.
2. For $m = 1$ to M:
    a. For $i = 1,2, \dots, N$, compute the residual
    $$\gamma_{\text{im}} = -\left[\frac{\partial L\big(y_i, f(x_i)\big)}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$$
    b. Fit a regression tree to the targets γ_{im} resulting in terminal regions $R_{mj}, j = 1,2, \dots, J_m$.
    c. For $j = 1,2, \dots, J_m$, compute
    $$c_{mj} = lr \times argmin_c \sum_{x_i \in R_{mj}} L(y_i, f_{m-1}(x_i) + c)$$
    d. Update $\mathrm{f_m}(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \mathrm{c}_{mj} I\big(x \in R_{mj}\big)$
3. The regression tree is $\hat{f}(x) = \mathrm{f_M}(x) = f_0(x) + \sum_{m=1}^{M} \sum_{j=1}^{J_m} \mathrm{c}_{mj} I\big(x \in R_{mj}\big)$

Implementation details:

You are required to implement the methods "`fit(self, X, y)`" and "`predict(self, X)`" in the file "GradientBoostingRegressor.py". You are not allowed to modify the definition of the given attributes and methods, i.e. no name change and no additional parameters in the methods. But you can add other methods or attributes in the class for your convenience.

In the function "`fit(self, X, y)`", you should update the `self.estimators`, whose type is nump array. Each element in this array is a regression tree object.

4. Hierarchical Clustering Algorithm

In this part, you are going to implement an Agglomerative Hierarchical Clustering Algorithm. We have provided the data and the test cases to you. You should use **Python 3.X** in your implementation.

The algorithm of Agglomerative Hierarchical Clustering is summarized as follows:

Input: Data points $X = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$; $x_i \in \mathbf{R}$ *and* $y_i \in \mathbf{R}$ are the coordinates.

Output: Clustering history: a list of pairs of the cluster ID, $H = \{(i_1, j_1), (i_2, j_2), \dots\}$, that indicates which pair of clusters are merged first**;** for example, $\{(1,3), (2,4), \dots\}$ indicates that (C_1, C_3) are merged first, then (C_2, C_4) are merged, ...

Steps:

1. $C_i \leftarrow \{(x_i, y_i)\}$, for i $\in \{1, \dots, N\}$, # *current clusters* $\leftarrow N$
2. Compute $ProximityMatrix$[i, j], for i, j $\in \{1, \dots, N\}$
3. $ClusterIndexSet = \{1, \dots, N\}$; $H = []$
4. Repeat:

   Find (p, q) with $\underset{i,j \in \text{ClusterIndexSet}}{\text{argmin}}$ $ProximityMatrix$[i, j]

   Merge (C_p, C_q) together as C_{N+1}

   Update $ProximityMatrix[i, N+1]$, for each i $\in$ ClusterIndexSet

   Append (p, q) into H

   $N \leftarrow N + 1$

   Remove p from $ClusterIndexSet$

   Remove q from $ClusterIndexSet$

   Insert N into $ClusterIndexSet$

   Until $sizeof(ClusterIndexSet) = 1$
5. Return H

In the algorithm above, you are required to implement both the single linkage and the complete linkage as the proximity definitions. You could print the clustering history out, which would be useful to find out the difference between the single linkage and the complete linkage.

Implementation details:

You are required to implement the function "`single_linkage(points, p, q)`", "`complete_linkage(points, p, q)`", and the class "`MyAgglomerativeClustering()`" in the file "Clustering.py". In this class, you are required to implement the method "`find_clusters_to_merge(self)`", "`merge_cluster(self, p, q)`", and "`update_proximity(self, new_cluster)`". You can add other methods or attributes in the class for your convenience.

The clustering history of your algorithm will be automatically recorded by "`MyAgglomerativeClustering().fit(X)`".

6. Library

You can use Numpy in your implementation. But DO NOT use external machine learning libraries like scipy, scikit-learn in your implementation. DO NOT copy the code from the internet, e.g. Github.

7. Cooperation

It is okay for you to discuss implementation ideas with your classmates. However, you need to write the code yourself, plagiarism will not be tolerated in this module.

8. Error Reporting

If you spot any bug in the assignment, please report it in the forum.

9. Marking Criteria

We will generate several test cases to test the correctness of your code.

For Decision Tree Regressor and Gradient Boosting Regressor, there are 3 test cases. For each test case,

- Decision tree model: 1 point
- Decision tree prediction result: 1 point
- Gradient boosting model: 1 point
- Gradient boosting prediction result: 1 point

You will also get some points from the implementation (code) for Decision Tree Regressor and Gradient Boosting Regressor even if you cannot pass all test cases. There are 4 points in total.

For Hierarchical Clustering Algorithm, there are also 3 test cases. For each test case,

- Single linkage result: 1 point
- Complete linkage result: 1 point

You will also get some point from the implementation (code) for Hierarchical Clustering Algorithm even if you cannot pass all test cases. There are 3 points for it.

If the submission is not following the format below, 1 point will be deducted.

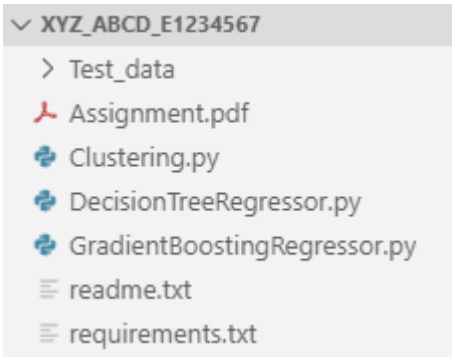
If the code needs simple (trivial) modification for running due to careless mistakes, points will be deducted depending on the lines of modification.

If you miss the submission deadline, 15% of the mark will be deducted per day late (17:01 is the start of one day).

10. Submission Format

When you submit your code, please zip your assignment folder and name your folder in the following format "YourNameInLumiNUS_YourUserIDInLumiNUS.zip" (You can check the displayed information from the Class & Groups tab in the module CS5228 from LumiNUS). An example may be "XYZ_ABCD_E1234567.zip". In addition, you need to make sure that if we unzip your folder, the files “DecisionTreeRegressor.py”, “GradientBoostingRegressor.py”, and “Clustering.py” reside in the "XYZ_ABCD_E1234567" folder.

Below is a sample submission format for your reference.



Please follow the submission format tightly, as we will apply this format to automatically mark your code. So, if you fail to obey this format, we will not be able to mark your code and corresponding marks will be deducted from you.

Appendix: Instructions of Using the Test Data

Our TAs have implemented all models and generated test cases in Test_data folder. You can compare your output files and the files generated by TA's implementation to check if your implementation is correct or not. Note that after submission, we will run other test cases for each model.

For Decision Tree Regressor and Gradient Boosting Regressor,

x_0.csv is input feature file.
y_0.csv is the input label file.

decision_tree_0_0.json and decision_tree_0_1.json are generated by the Decision Tree Regressor (implemented by TA) with x_0.csv and y_0.csv as the inputs; You can compare your output files and these two files to check if your implementation is correct or not.

gradient_boosting_0_0.json and gradient_boosting_0_1.json are generated by the Gradient Boosting Regressor (implemented by TA) with x_0.csv and y_0.csv. as the inputs for you to test your implementation.

y_pred_decision_tree_0_0.csv and y_pred_decision_tree_0_1.csv are predictions made by decision_tree_0_0.json and decision_tree_0_1.json with x_0.csv as the input.

For Agglomerative Hierarchical Clustering,

clustering_0.csv is the input data file.

clustering_single_0.csv is the history H of the single linkage agglomerative clustering for the inputs in clustering_0.csv. It is generated by TA's implementation.

Likewise, clustering_complete_0.csv is the correct history H of the complete linkage agglomerative clustering for the same inputs.