gbdt

March 6, 2024

[1]:
```
from multiprocessing import Pool
from functools import partial
import numpy as np
from numba import jit
```

[2]:
```
#TODO: loss of least square regression and binary logistic regression
'''
    pred() takes GBDT/RF outputs, i.e., the "score", as its inputs, and returns␣
 ↪predictions.
    g() is the gradient/1st order derivative, which takes true values "true"␣
 ↪and scores as input, and returns gradient.
    h() is the heassian/2nd order derivative, which takes true values "true"␣
 ↪and scores as input, and returns hessian.
'''
class loss(object):
    '''Loss class for mse. As for mse, pred function is pred=score.'''
    def pred(self,X_test,trees,log_threshold):
        return 0

    def g(self, y_true, pred):
        return 0

    def h(self, pred):
        return 0

class leastsquare(loss):
    '''Loss class for mse. As for mse, pred function is pred=score.'''
    def pred(self,X_test,trees,log_threshold):
        predictions = []
        #print("Tree Length : " + str(len(trees)))
        for test in X_test:
            #count = 0
            score = 0.0
            for tree in trees:
                curr_node = tree.root_node
                while curr_node.is_leaf is not True  :
                    curr_node = curr_node.forward(test)
                #print(str(count) + " " + str(curr_node.weight))
                #count = count + 1
                score = score + curr_node.weight
            score = score/len(trees)
            predictions.append(score)
        return predictions

    def g(self, y_true, pred):
        return (-2*(y_true - pred))

    def h(self, y_true, pred):
        return 2*np.ones(y_true.shape)

class logistic(loss):
    '''Loss class for log loss. As for log loss, pred function is logistic␣
 ↪transformation.'''
    def pred(self,X_test,trees,log_threshold):
        predictions = []
        #print(len(trees))
        for test in X_test:
            temp = np.zeros(len(trees))
            for i in range(len(trees)):
                tree = trees[i]
                curr_node = tree.root_node
                while(curr_node.is_leaf == False):
                    curr_node = curr_node.forward(test)
                temp[i] = curr_node.weight
            min = np.min(temp)
            max = np.max(temp)
            #print(min)
            #print(max)
            if max != min :
                temp = (temp - min)/(max - min)
            temp = np.rint(temp)
            unique, counts = np.unique(temp, return_counts=True)
            #print(unique)
            if(unique.size > 1):
                if counts[1] > counts[0] :
                    predictions.append(1.0)
                else :
                    predictions.append(0.0)
            else :
                predictions.append(unique[0])
        return predictions

    def g(self, y_true, pred):
        return ((((1-y_true)*np.exp(pred)) - y_true)/(np.exp(pred) + 1))
    def h(self, y_true, pred):
        return (np.exp(pred)/((np.exp(pred) + 1)*(np.exp(pred) + 1)))*np.
 ↪ones(y_true.shape)
```

[3]:
```
# TODO: class of a node on a tree
class TreeNode(object):
    '''
    Data structure that are used for storing a node on a tree.

    A tree is presented by a set of nested TreeNodes,
    with one TreeNode pointing two child TreeNodes,
    until a tree leaf is reached.

    A node on a tree can be either a leaf node or a non-leaf node.
    '''

    #TODO
    def __init__(self, X1, X2, index_y, threshold, value_y, is_leaf):
        self.is_leaf = is_leaf
        self.left_child = X1
        self.right_child = X2
        self.feature_index = index_y
        self.threshold = threshold
        self.weight = value_y

    def forward(self, x):
        #print(self.weight)
        #print(self.feature_index)
        #print(self.threshold)
        #print(self.is_leaf)
        #print(self.left_child.is_leaf)
        #print(self.right_child.is_leaf)
        if self.is_leaf == True :
            raise RuntimeError('This is a leaf node and you cannot forward from␣
 ↪it')
        elif x[self.feature_index] < self.threshold:
            #print(self.left_child.is_leaf)
            return self.left_child
        else :
            #print(self.right_child.is_leaf)
            return self.right_child
```

[4]:
```
# TODO: class of single tree
class Tree(object):
    '''
    Class of a single decision tree in GBDT
    Parameters:
        n_threads: The number of threads used for fitting and predicting.
        max_depth: The maximum depth of the tree.
        min_sample_split: The minimum number of samples required to further␣
↪split a node.
        lamda: The regularization coefficient for leaf prediction, also known␣
↪as lambda.
        gamma: The regularization coefficient for number of TreeNode, also know␣
↪as gamma.
        rf: rf*m is the size of random subset of features, from which we select␣
↪the best decision rule,
            rf = 0 means we are training a GBDT.
    '''

    def __init__(self, root_node = None, n_threads = None,
                 max_depth = 3, min_sample_split = 10,
                 lamda = 1, gamma = 0, pred = 0):
        self.n_threads = n_threads
        self.max_depth = max_depth
        self.min_sample_split = min_sample_split
        self.root_node = root_node
        self.lamda = lamda
        self.gamma = gamma
        self.pred = pred
        self.treenodes = []

    def fit(self, X, y, loss):
        '''
        train is the training data matrix, and must be numpy array (an n_train␣
↪x m matrix).
        g and h are gradient and hessian respectively.
        '''
        g = loss.g(y, self.pred)
        h = loss.h(y, self.pred)
        #print(g)
        #print(h)
        self.root_node = self.construct_tree(X, y, g, h, 0)
        #print(self.root_node.weight)
        return self

    def split_dataset(self, X, y, feature_index, threshold):
        left_mask = X[:, feature_index] < threshold
        right_mask = X[:, feature_index] >= threshold
        return X[left_mask], X[right_mask], y[left_mask], y[right_mask]

    def calculate_gain(self, g_l, h_l, g_r, h_r):
        G_l_tot = g_l.sum()
        G_r_tot = g_r.sum()
        H_l_tot = h_l.sum()
        H_r_tot = h_r.sum()
        gain = (0.5)*(((G_l_tot)*(G_l_tot)/(H_l_tot + self.lamda)) +␣
↪((G_r_tot)*(G_r_tot)/(H_r_tot + self.lamda)) -␣
↪(((G_l_tot+G_r_tot)*(G_l_tot+G_r_tot))/(H_l_tot + H_r_tot+ self.lamda))) -␣
↪self.gamma
        return gain

    def find_threshold(self, g, h, X, y, feature_index):
        '''
        Given a particular feature $p_j$,
        return the best split threshold $\tau_j$ together with the gain that is␣
↪achieved.
        '''
        best_gain = 0
        best_threshold = 0
        thresholds = np.unique(X[:, feature_index])
        for threshold in thresholds:
            X_left, X_right, y_left, y_right = self.split_dataset(X, y,␣
↪feature_index, threshold)
            if len(y_left) == 0 or len(y_right) == 0:
                continue
            left_mask = X[:, feature_index] < threshold
            right_mask = X[:, feature_index] >= threshold
            gain = self.calculate_gain(g[left_mask], h[left_mask],␣
↪g[right_mask], h[right_mask])
            #print("Gain : " +str(gain))
            if gain > best_gain:
                best_gain = gain
                best_threshold = threshold
        return [threshold, best_gain]

    def construct_tree(self, X, y, g, h, current_depth):
        '''
        Node Addition, which is recursively used to grow a tree.
        First we should check if we should stop further splitting.

        The stopping conditions include:
            1. tree reaches max_depth $d_{max}$
            2. The number of sample points at current node is less than␣
↪min_sample_split, i.e., $n_{min}$
            3. gain <= 0
        '''
        if current_depth > self.max_depth or y.size < self.min_sample_split :
            return None
        best_feature, threshold, best_gain = self.find_best_decision_rule(X, y,␣
↪g, h)
        #print("best_feature : " + str(best_feature))
        #print("threshold : " + str(threshold))
        #print("best_gain : " + str(best_gain))
        if best_gain == 0 :
            return None
        #print(X[:, best_feature])
        left_mask = X[:, best_feature] < threshold
        right_mask = X[:, best_feature] >= threshold
        node_left = self.construct_tree(X[left_mask], y[left_mask],␣
↪g[left_mask], h[left_mask], current_depth + 1)
        node_right = self.construct_tree(X[right_mask], y[right_mask],␣
↪g[right_mask], h[right_mask], current_depth + 1)
        h_val = h.sum()
        weight = -(g.sum()/(h_val + self.lamda))
        is_leaf = False
        if node_left is None and node_right is not None :
            node_left = TreeNode(None, None, 0, 0, weight, True)
        elif node_right is None and node_left is not None :
            node_right = TreeNode(None, None, 0, 0, weight, True)
        elif node_right is None and node_left is None :
            is_leaf = True
        node = TreeNode(node_left, node_right, best_feature, threshold, weight,␣
↪is_leaf)
        self.treenodes.append(node)
        #print(node)
        return node

    def find_best_decision_rule(self, X, y, g, h):
        '''
        Return the best decision rule [feature, treshold], i.e., $(p_j,␣
↪\tau_j)$ on a node j,
        train is the training data assigned to node j
        g and h are the corresponding 1st and 2nd derivatives for each data␣
↪point in train
        g and h should be vectors of the same length as the number of data␣
↪points in train

        for each feature, we find the best threshold by find_threshold(),
        a [threshold, best_gain] list is returned for each feature.
        Then we select the feature with the largest best_gain,
        and return the best decision rule [feature, treshold] together with its␣
↪gain.
        '''
        best_gain = 0
        best_threshold = 0
        best_feature = 0
        for feature_index in range(X.shape[1]):
            #print("Feature " + str(feature_index))
            threshold, gain = self.find_threshold(g, h, X, y, feature_index)
            if gain > best_gain:
                best_gain = gain
                best_threshold = threshold
                best_feature = feature_index
        return best_feature, threshold, best_gain
```

[5]:

```
# TODO: class of Random Forest
class RF(object):
    '''
    Class of Random Forest

    Parameters:
        n_threads: The number of threads used for fitting and predicting.
        loss: Loss function for gradient boosting.
            'mse' for regression task and 'log' for classfication task.
            A child class of the loss class could be passed to implement␣
↪customized loss.
        max_depth: The maximum depth d_max of a tree.
        min_sample_split: The minimum number of samples required to further␣
↪split a node.
        lamda: The regularization coefficient for leaf score, also known as␣
↪lambda.
        gamma: The regularization coefficient for number of tree nodes, also␣
↪know as gamma.
        rf: rf*m is the size of random subset of features, from which we select␣
↪the best decision rule.
        num_trees: Number of trees.
    '''
    def __init__(self,log_threshold = 0.5,
        n_threads = 0, loss = 'mse',
        max_depth = 3, min_sample_split = 10,
        lamda = 1, gamma = 0,
        rf = 0.99, num_trees = 100):
        self.log_threshold = log_threshold
        self.n_threads = n_threads
        if loss == 'mse':
            self.loss = leastsquare()
        else :
            self.loss = logistic()
        self.max_depth = max_depth
        self.min_sample_split = min_sample_split
        self.lamda = lamda
        self.gamma = gamma
        self.rf = rf
        self.num_trees = num_trees
        self.trees = []

    def fit(self, X, y):
        # X is n x m 2d numpy array
        # y is n-dim 1d array
        num_samples, num_features = X.shape
        for i in range(self.num_trees):
            #print ("Tree : " + str(i))
            m_rf = (int)(np.ceil(self.rf*num_features))
            m_rf_indices = np.random.choice(num_features, m_rf, replace=False)
            n_rf_indices = np.random.choice(num_samples, num_samples,␣
↪replace=True)
            XTree = X[n_rf_indices][:,m_rf_indices]
            #print(XTree.shape)
            yTree = y[n_rf_indices]
            prediction = 0
            tree = Tree(None, self.n_threads, self.max_depth, self.
↪min_sample_split,self.lamda, self.gamma, prediction)
            tree = tree.fit(XTree, yTree, self.loss)
            self.trees.append(tree)
        return self

    # test is the input to the model
    def predict(self, X_test):
        return self.loss.pred(X_test, self.trees, self.log_threshold)
```

[6]:
```
# TODO: class of GBDT
class GBDT(object):
    '''
    Class of gradient boosting decision tree (GBDT)

    Parameters:
        n_threads: The number of threads used for fitting and predicting.
        loss: Loss function for gradient boosting.
            'mse' for regression task and 'log' for classfication task.
            A child class of the loss class could be passed to implement␣
↪customized loss.
        max_depth: The maximum depth D_max of a tree.
        min_sample_split: The minimum number of samples required to further␣
↪split a node.
        lamda: The regularization coefficient for leaf score, also known as␣
↪lambda.
        gamma: The regularization coefficient for number of tree nodes, also␣
↪know as gamma.
```

```
        learning_rate: The learning rate eta of GBDT.
        num_trees: Number of trees.
    '''
    def __init__(self, log_threshold = 0.5,
        learning_rate=0.5, n_threads = 0, loss = 'mse',
        max_depth = 3, min_sample_split = 10,
        lamda = 1, gamma = 0,
        rf = 0.99, num_trees = 100):
        self.loss_type = loss
        self.n_threads = n_threads
        if loss == 'mse':
            self.loss = leastsquare()
        else :
            self.loss = logistic()
        self.max_depth = max_depth
        self.log_threshold = log_threshold
        self.min_sample_split = min_sample_split
        self.lamda = lamda
        self.gamma = gamma
        self.rf = rf
        self.num_trees = num_trees
        self.learning_rate = learning_rate
        self.trees = []

    def fit(self, X, y):
        # X is n x m 2d numpy array
        # y is n-dim 1d array
        num_samples, num_features = X.shape
        prediction = np.zeros(y.shape)
        for i in range(self.num_trees):
            #print ("Tree : " + str(i))
            tree = Tree(None, self.n_threads, self.max_depth, self.
↪min_sample_split,self.lamda, self.gamma, prediction)
            tree = tree.fit(X, y, self.loss)
            self.trees.append(tree)

            if self.loss_type == 'mse':
                new_pred =self.loss.pred(X, [tree], self.log_threshold)
            else :
                count0 = 0
                count1 = 1
                curr_node = tree.root_node
                new_pred = []
                for test in X:
                    while(curr_node.is_leaf == False):
                        curr_node = curr_node.forward(test)
                    if curr_node.weight > self.log_threshold:
```

```
                    count1 = count1 + 1
                else :
                    count0 = count0 + 1
                if count1 > count0 :
                    new_pred.append(1.0)
                else :
                    new_pred.append(0.0)
            prediction = self.learning_rate*prediction + np.array(new_pred)
        return self

    # test is the input to the model
    def predict(self, X_test):
        return self.loss.pred(X_test, self.trees, self.log_threshold)
```

[7]:
```
# TODO: Evaluation functions (you can use code from previous homeworks)

# RMSE
def root_mean_square_error(pred, y):
    diff_matrix = y - pred
    #print(y)
    #print(diff_matrix)
    rmse = diff_matrix**2
    rmse = rmse.sum()
    rmse = rmse/np.size(y)
    rmse = np.sqrt(rmse)
    return rmse

# precision
def accuracy(pred, y):
    dataset_size = y.size
    correct_predictions = 0
    #print(pred)
    #print(y)
    for i in range(dataset_size):
        if y[i] == (pred[i]):
            correct_predictions += 1
    precision = (correct_predictions/dataset_size)*100
    return precision
```

[8]:
```
# Run Random Forest Code
from sklearn import datasets
boston = datasets.load_boston()
X = boston.data
y = boston.target

# train-test split
from sklearn.model_selection import train_test_split
```

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=8)
RF_regression = RF(0.5, n_threads = 0, loss = 'mse', max_depth = 8,␣
 ↪min_sample_split = 10, lamda = 1, gamma = 1,rf = 0.8, num_trees = 99)
RF_regression = RF_regression.fit(np.array(X_train), np.array(y_train))
pred_train = RF_regression.predict(X_train)
pred_test = RF_regression.predict(X_test)
rmse_test = root_mean_square_error(pred_test, y_test)
rmse_train = root_mean_square_error(pred_train, y_train)
print("Training RMSE of the Random Forest Method for the Boston Dataset is : ",␣
 ↪rmse_train)
print("Testing RMSE of the Random Forest Method for the Boston Dataset is : ",␣
 ↪rmse_test)
print("Training Set RMSE of Linear Regression for Boston Dataset (from HW2): 4.
 ↪820626531838223")
print("Testing Set RMSE of Linear Regression for Boston Dataset (from HW2): 5.
 ↪209217510530916")
print("Training Set RMSE of Ridge Regression for Boston Dataset (from HW2): 4.
 ↪829777333975097")
print("Testing Set RMSE of Ridge Regression for Boston Dataset (from HW2): 5.
 ↪189347305423606")
```

```
/Users/prateek/anaconda3/envs/AnacondaTest/lib/python3.11/site-
packages/sklearn/utils/deprecation.py:87: FutureWarning: Function load_boston is
deprecated; `load_boston` is deprecated in 1.0 and will be removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np

        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::
        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.
  warnings.warn(msg, category=FutureWarning)
```

```
Training RMSE of the Random Forest Method for the Boston Dataset is :
9.229962989283528
Testing RMSE of the Random Forest Method for the Boston Dataset is :
8.975222658557659
Training Set RMSE of Linear Regression for Boston Dataset (from HW2):
4.820626531838223
Testing Set RMSE of Linear Regression for Boston Dataset (from HW2):
5.209217510530916
Training Set RMSE of Ridge Regression for Boston Dataset (from HW2):
4.829777333975097
Testing Set RMSE of Ridge Regression for Boston Dataset (from HW2):
5.189347305423606
```

[20]:
```python
# load data
from sklearn.datasets import fetch_openml
X, y = fetch_openml('credit-g', version=1, return_X_y=True, data_home='credit/
 ↪',as_frame=False)
y = np.array(list(map(lambda x: 1 if x == 'good' else 0, y)))

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=8)
#print(X_train)
RF_regression = RF(log_threshold = 0.5,n_threads = 0, loss = 'class', max_depth␣
 ↪= 8, min_sample_split = 10, lamda = 1, gamma = 1,rf = 0.5, num_trees = 99)
RF_regression = RF_regression.fit(np.array(X_train), np.array(y_train))
pred_train = RF_regression.predict(X_train)
pred_test = RF_regression.predict(X_test)
acc_test = accuracy(pred_test, y_test)
acc_train = accuracy(pred_train, y_train)
print("Training Accuracy of the Random Forest Method for the Credit-g Dataset␣
 ↪is : ", acc_train)
print("Testing Accuracy of the Random Forest Method for the Credit-g Dataset is␣
 ↪: ", acc_test)
```

```
Training Accuracy of the Random Forest Method for the Credit-g Dataset is :
69.71428571428572
Testing Accuracy of the Random Forest Method for the Credit-g Dataset is :  70.0
```

[10]:
```
# load data
from sklearn import datasets
breast_cancer = datasets.load_breast_cancer()
X = breast_cancer.data
y = breast_cancer.target

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=8)
RF_regression = RF(log_threshold = 0.5, n_threads = 0, loss = 'class',␣
 ↪max_depth = 10, min_sample_split = 10, lamda = 1, gamma = 1,rf = 0.5,␣
 ↪num_trees = 99)
RF_regression = RF_regression.fit(np.array(X_train), np.array(y_train))
pred_train = RF_regression.predict(X_train)
pred_test = RF_regression.predict(X_test)
acc_test = accuracy(pred_test, y_test)
acc_train = accuracy(pred_train, y_train)
print("Training Accuracy of the Random Forest Method for the Breast Cancer␣
 ↪Dataset is : ", acc_train)
print("Testing Accuracy of the Random Forest Method for the Breast Cancer␣
 ↪Dataset is : ", acc_test)
```

```
Training Accuracy of the Random Forest Method for the Breast Cancer Dataset is :
63.31658291457286
Testing Accuracy of the Random Forest Method for the Breast Cancer Dataset is :
61.40350877192983
```

[11]:
```
# TODO: GBDT regression on boston house price dataset

# load data
from sklearn import datasets
boston = datasets.load_boston()
X = boston.data
y = boston.target

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=8)
learning_rate = 0.2
```

```
GBDT_regression = GBDT(0.5, learning_rate, n_threads = 0, loss = 'mse',␣
  ↪max_depth = 8, min_sample_split = 10, lamda = 1, gamma = 1,rf = 0.5,␣
  ↪num_trees = 99)
GBDT_regression = GBDT_regression.fit(np.array(X_train), np.array(y_train))
pred_train = GBDT_regression.predict(X_train)
pred_test = GBDT_regression.predict(X_test)
rmse_test = root_mean_square_error(pred_test, y_test)
rmse_train = root_mean_square_error(pred_train, y_train)
print("Training RMSE of the GBDT Method for the Boston Dataset is : ",␣
  ↪rmse_train)
print("Testing RMSE of the GBDT Method for the Boston Dataset is : ", rmse_test)
```

```
Training RMSE of the GBDT Method for the Boston Dataset is :  15.3444415615304
Testing RMSE of the GBDT Method for the Boston Dataset is :  15.524724610785293
```

[12]:
```
# TODO: GBDT classification on credit-g dataset

# load data
from sklearn.datasets import fetch_openml
X, y = fetch_openml('credit-g', version=1, return_X_y=True, data_home='credit/
  ↪',as_frame=False)
y = np.array(list(map(lambda x: 1 if x == 'good' else 0, y)))

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
  ↪random_state=8)
learning_rate = 0.9
GBDT_regression = GBDT(0.5, learning_rate, n_threads = 0, loss = 'class',␣
  ↪max_depth = 8, min_sample_split = 10, lamda = 5, gamma = 0.5,rf = 0.5,␣
  ↪num_trees = 100)
GBDT_regression = GBDT_regression.fit(np.array(X_train), np.array(y_train))
pred_train = GBDT_regression.predict(X_train)
pred_test = GBDT_regression.predict(X_test)
acc_test = accuracy(pred_test, y_test)
acc_train = accuracy(pred_train, y_train)
print("Training Accuracy of the GBDT Method for the Credit-g Dataset is : ",␣
  ↪acc_train)
print("Testing Accuracy of the GBDT Method for the Credit-g Dataset is : ",␣
  ↪acc_test)
```

```
Training Accuracy of the GBDT Method for the Credit-g Dataset is :
68.85714285714286
Testing Accuracy of the GBDT Method for the Credit-g Dataset is :
71.33333333333334
```

[13]:
```
# TODO: GBDT classification on breast cancer dataset

# load data
from sklearn import datasets
breast_cancer = datasets.load_breast_cancer()
X = breast_cancer.data
y = breast_cancer.target

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=8)

# train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=8)
learning_rate = 0.8
GBDT_regression = GBDT(0.5, learning_rate, n_threads = 0, loss = 'class',␣
 ↪max_depth = 4, min_sample_split = 10, lamda = 6, gamma = 0.8,rf = 0.5,␣
 ↪num_trees = 60)
GBDT_regression = GBDT_regression.fit(np.array(X_train), np.array(y_train))
pred_train = GBDT_regression.predict(X_train)
pred_test = GBDT_regression.predict(X_test)
acc_test = accuracy(pred_test, y_test)
acc_train = accuracy(pred_train, y_train)
print("Training Accuracy of the GBDT Method for the Breast Cancer Dataset is :␣
 ↪", acc_train)
print("Testing Accuracy of the GBDT Method for the Breast Cancer Dataset is :␣
 ↪", acc_test)
```

```
Training Accuracy of the GBDT Method for the Breast Cancer Dataset is :
63.31658291457286
Testing Accuracy of the GBDT Method for the Breast Cancer Dataset is :
61.40350877192983
```

[ ]:
```

```