In [245]:
```
#from multiprocessing import Pool
#from functools import partial
import numpy as np
#from numba import jit
```

loss of least square regression:

$$(y_i - \hat{y}_i)^2$$

gradient =

$$2(\hat{y}_i - y_i)$$

loss in logistic:

$$l(y, \hat{y}) = y \log(1 + \exp(-\hat{y})) + (1 - y) \log(1 + \exp(\hat{y}))$$

gradient of log_loss:

$$g = 1/(1 + \exp(-\hat{y})) - y$$

hessian of log_loss:

$$h = 1/(1 + \exp(-\hat{y})) * (1 - 1/(1 + \exp(-\hat{y})))$$

In [246]:
```
#TODO: loss of least square regression and binary logistic regress
'''
    pred() takes GBDT/RF outputs, i.e., the "score", as its inputs
    g() is the gradient/1st order derivative, which takes true val
    h() is the heassian/2nd order derivative, which takes true val
'''
class leastsquare(object):
    '''Loss class for mse. As for mse, pred function is pred=score
    def pred(self,score):
        return score

    def g(self,true,score):
        return 2*(score - true)

    def h(self,true,score):
#         return 2
        return np.ones_like(score) * 2

class logistic(object):
    '''Loss class for log loss. As for log loss, pred function is
    def pred(self,score):
        return 1 / (1 + np.exp(-score))


    def g(self,true,score):
        pred = self.pred(score)
        return (pred - true)

    def h(self,true,score):
        pred = self.pred(score)
        return pred * (1 - pred)
```

In [247]:

```
# 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, depth = None, split_feature = None, split_t

#         # store essential information in every tree node
#         self.X = X #features associated with node
#         self.y = y
        self.depth = depth
        self.is_leaf = left_child is None and right_child is None
        self.split_feature = split_feature
        self.split_threshold = split_threshold
        self.left_child = left_child
        self.right_child = right_child
        self.prediction_value = prediction_value
#         self.prediction_probs = None
```

In [248]:

```
# 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 pred
        max_depth: The maximum depth of the tree.
        min_sample_split: The minimum number of samples required t
        lamda: The regularization coefficient for leaf prediction,
        gamma: The regularization coefficient for number of TreeNo
        rf: rf*m is the size of random subset of features, from wh
            rf = 0 means we are training a GBDT.
    '''

    def __init__(self, n_threads = None,
                 max_depth = 3, min_sample_split = 10,
                 lamda = 1, gamma = 0, rf = 0):

        self.n_threads = n_threads
        self.max_depth = max_depth
        self.min_sample_split = min_sample_split
        self.lamda = lamda
        self.gamma = gamma
        self.rf = 0
        self.int_member = 0

    def fit(self, train, g, h):
        '''
        train is the training data matrix, and must be numpy array
        g and h are gradient and hessian respectively.
        '''
        #TODO
#         if g is None:
#             g = np.zeros_like(train[:, -1])
#         if h is None:
#             h = np.zeros_like(train[:, -1])

        self.tree = self.construct_tree(train, g, h, self.max_dept
#         self.int_member = np.mean(train[:, -1])

        return self

    def predict(self, test):
        '''
        test is the test data matrix, and must be numpy arrays (an
        Return predictions (scores) as an array.
        '''
        result = []
        num_samples = test.shape[0]

        for i in range(num_samples):
            instance = test[i]
            node = self.tree
            while node.is_leaf is False:
                node = node.left_child if instance[node.split_feat
            result.append(node.prediction_value)
        return np.array(result)
```

```python
    def construct_tree(self, train, g, h, max_depth):
#         prediction_value = np.mean(train[:, -1])  # Calculate th

        num_samples = train.shape[0]
#         total_gradient, total_hessian = np.sum(g), np.sum(h)
        sum_g = np.sum(g)
        sum_h = np.sum(h)
        prediction_value = -sum_g / (sum_h + self.lamda)


        #TODO
        if train.shape[0] < self.min_sample_split or max_depth <=
            return TreeNode(prediction_value=prediction_value)


        feature, threshold, gain = self.find_best_decision_rule(tr

        if gain <= 0:
            return TreeNode(prediction_value=prediction_value)

        if threshold is None:
            raise ValueError("Threshold cannot be None")

        #split node
        left_indices = train[:, feature] < threshold
        right_indices = ~left_indices

        #recursively apply construct_tree function until above con
        left_child = self.construct_tree(train[left_indices], g[le
        right_child = self.construct_tree(train[right_indices], g[

        return TreeNode(split_feature = feature, split_threshold =
                        left_child = left_child, right_child = rig

    def find_best_decision_rule(self, train, g, h):
        '''
        Return the best decision rule [feature, treshold], i.e., $
        train is the training data assigned to node j
        g and h are the corresponding 1st and 2nd derivatives for
        g and h should be vectors of the same length as the number

        for each feature, we find the best threshold by find_thres
        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] toge
        '''
        #TODO 写了
        best_gain = -float('inf')
        best_feature = 0
        best_threshold = 0

        for feature in range(train.shape[1]): #for X
            threshold, gain = self.find_threshold(g, h, train[:, f
            if gain > best_gain:
                best_gain = gain
                best_feature = feature
                best_threshold = threshold

        return best_feature, best_threshold, best_gain

    def find_threshold(self, g, h, train):
```

```
        '''
        Given a particular feature $p_j$,
        return the best split threshold $\tau_j$ together with the
        '''
        #TODO
        num = train.shape[0] #number of samples
        sorted_indices = np.argsort(train)
        sorted_values = train[sorted_indices]
        best_threshold = 0
        best_gain = -float('inf') # initialize gain with small num
        sorted_g = g[sorted_indices]
        sorted_h = h[sorted_indices]

        sum_g = np.sum(sorted_g)
        sum_h = np.sum(sorted_h)
        l_g, l_h, r_g, r_h = 0, 0, sum_g, sum_h

        for i in range(1, num):

            l_g += sorted_g[i - 1]
            r_g -= sorted_g[i - 1]
            l_h += sorted_h[i - 1]
            r_h -= sorted_h[i - 1]

            if sorted_values[i] != sorted_values[i - 1]:
                gain = 0.5 * ((l_g ** 2 / (l_h + self.lamda)) + (r

                if gain > best_gain:
                    best_threshold = (sorted_values[i] + sorted_va
                    best_gain = gain


        return [best_threshold, best_gain]
```

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.

In [249]:

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

    def __init__(self,
        n_threads = None, loss = 'mse',
        max_depth = 3, min_sample_split = 10,
        lamda = 1, gamma = 0,
        rf = 0.99, num_trees = 100):

        self.n_threads = n_threads
        self.loss = loss
        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 = []
#         self.avg_prediction = None

    def fit(self, train, target): # train is n x m 2d numpy array;

        #TODO
        #score is prediction
#         score = np.zeros((len(train), self.num_trees))
        n_samples, n_features = train.shape
        sub = int(self.rf * n_features)

        #loop through the length of target y
        for i in range(self.num_trees):
            #Bootstrap, select len(train) numbers from len(train)
            indices = np.random.choice(train.shape[0], train.shape
            f_indices = np.random.choice(n_features, sub, replace=

            boot_train = train[indices][:, f_indices] #X
            boot_target = target[indices] #y
            boot_train_target = np.column_stack((boot_train, boot_

            #create instance of Tree, fit with bootstraped data
            tree = Tree(max_depth=self.max_depth, min_sample_split
            tree.fit(boot_train_target, np.full(boot_train.shape[0

#             score += tree.predict(train)
#             score[:, i] = tree.predict(train)
            self.trees.append((tree, f_indices))


#         return self

    def predict(self, test):
        #TODO
#         scores = np.zeros((test.shape[0], len(self.trees)))

#         # Make predictions using each tree
#         for i, tree in enumerate(self.trees):
#             scores[:, i] = tree.predict(test)
#         avg_prediction = np.mean(scores, axis=1)

        if not self.trees:
            raise ValueError("tree is none.")
```

```
        predictions = np.array([tree.predict(test[:, f_indices]) f
        if self.loss == 'mse':
            final_predictions = np.mean(predictions, axis=0)
        return final_predictions
#         return avg_prediction
```

GBDT: gradient-based method, lr required

Random Forest: ensemble method, combines the predictions from multiple decision trees. Each tree trained independently.

In [251]:

```
# TODO: class of GBDT
class GBDT(object):

    def __init__(self,
        n_threads = None, loss = 'mse',
        max_depth = 3, min_sample_split = 10,
        lamda = 1, gamma = 0,
        learning_rate = 0.1, num_trees = 100):

        self.n_threads = n_threads
        self.loss = loss
        if loss == 'mse':
            self.loss = leastsquare()
        elif loss == 'log':
            self.loss = logistic()

        self.max_depth = max_depth
        self.min_sample_split = min_sample_split
        self.lamda = lamda
        self.gamma = gamma
        self.learning_rate = learning_rate
        self.num_trees = num_trees
        self.trees = []

    def fit(self, train, target):

        pred = np.full(target.shape[0], 0)

        g, h = self.learning_rate * self.loss.g(target, pred), sel

        for _ in range(self.num_trees):

            new_tree = Tree(n_threads = self.n_threads)
            new_tree.fit(train, g, h)

            self.trees.append(new_tree)

            pred = self.predict(train)
            g, h = self.learning_rate * self.loss.g(target, pred),
#             import pdb
#             pdb.set_trace()
        return self

    def predict(self, test):
        #TODO
#         import pdb
#         pdb.set_trace()
        predictions = np.array([tree.predict(test) for tree in sel
        score = np.sum(predictions, axis = 0)
        return self.loss.pred(score)
#     def fit(self, train, target):

#         score = np.full(target.shape[0], 0)
#         g = self.learning_rate * self.loss.g(target, score)
#         h = (self.learning_rate**2) * self.loss.h(target, score)

#         for _ in range(self.num_trees):

#             #create a tree instance of class Tree, fit Tree
#             tree = Tree(n_threads = self.n_threads)
```

```
#             tree.fit(train, g, h)

#             #update score(prediction), lr*current_score
#             score = tree.predict(train)
#             g = self.learning_rate * self.loss.g(target, score)
#             h = (self.learning_rate**2) * self.loss.h(target, sc

#             self.trees.append(tree)

#         return self

#     def predict(self, test):
#         #TODO
#         scores = np.zeros((test.shape[0],))
#         for tree in self.trees:
#             scores += self.learning_rate * tree.predict(test)

#         return self.loss.pred(scores)
```

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.

fit(self, train, g, h):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. **predict(self,test)**:test is the test data matrix, and must be numpy arrays (an n_test x m matrix). Return predictions (scores) as an array. **construct_tree**：Tree construction，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

find_best_decision_rule： Return the best decision rule [feature, treshold], i.e., (p_j, τ_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 traing 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.

find_threshold: Given a particular feature p_j, return the best split threshold τ_j together with the gain that is achieved.

In [252]:
```python
#写过了
#TODO: Evaluation functions (you can use code from previous homewo

# RMSE
def root_mean_square_error(pred, y):
    #TODO (1, 1, 1, 3, 4) -> (3, 4)
#     rmse = np.sqrt(np.mean((pred - y) ** 2))
#     return rmse
    return np.sqrt(np.mean((pred - y) ** 2))

# precision
def accuracy(pred, y):
#     #TODO

    # <0.5 to 0; > 0.5 to 1
    pred = np.round(pred)
    return np.mean(pred == y)
```

Q2.6

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

Answer: The computational complexity of optimized tree is $O(m \times n^d)$. At each node, we consider m features, and each feature contains n samples, we iterate over each node into a depth of d.

2. 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.

Answer: The most expensive computation in GBDT is arithmetic operation on gradient. One way to improve efficiency is setting a standard for early stopping. If the performance (gain) improvment does not reach this creteria, then the iteration stops early.

3. Which parts of GBDT training can be computed in parallel? Briefly describe your solution, and use it in your implementation

Answer: Finding the best split point at each node can be done in paralle; Parallelize the training of each tree: train different trees simultaneously subsets of the data by distributing the data across multiple processing units or nodes. Train on each tree, then combine to get the result.

Housing price dataset

The RMSE of DGBT and random forest for regression task indicates a better performance than linear regression and ridge regression models. And GBDT performs even better than the random forest regression model:

Train RMSE for RF is: 3.0152911495325685
Test RMSE for RF is: 4.114077453502243

Training RMSE for GBDT is: 1.7022012576754466
Testing RMSE for GBDT is: 3.3361789632748824

Recall (pasted from previous HW question answers):
Train RMSE for linear regression: 4.820626531838223
Train RMSE for ridge regression: 4.82636361174151

Train RMSE for linear regression: 5.209217510531067
Train RMSE for ridge regression: 5.191203625647021

In [253]:
```
# TODO: GBDT regression on boston house price dataset

# load data
import numpy as np
import pandas as pd

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


# 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_siz
print(X.shape, y.shape, X_train.shape, y_train.shape, X_test.shape
```

```
(506, 13) (506,) (354, 13) (354,) (152, 13) (152,)
```

In [254]:
```

gbdt = GBDT(num_trees = 100, learning_rate = 0.001, n_threads = 1)

gbdt.fit(X_train, y_train)

pred_train_gbdt = gbdt.predict(X_train)
pred_test_gbdt = gbdt.predict(X_test)

# Compute RMSE for training and testing
rmse_train_gbdt = root_mean_square_error(pred_train_gbdt, y_train)
rmse_test_gbdt = root_mean_square_error(pred_test_gbdt, y_test)

print("GBDT regression on boston house price dataset")
print("Training RMSE for GBDT is:", rmse_train_gbdt)
print("Testing RMSE for GBDT is:", rmse_test_gbdt)
```

```
GBDT regression on boston house price dataset
Training RMSE for GBDT is: 1.7022012576754466
Testing RMSE for GBDT is: 3.3361789632748824
```

In [255]:
```
# double check with sklearn

from sklearn.ensemble import GradientBoostingRegressor as GBDT

gb_regressor = GBDT(n_estimators=100, learning_rate=0.1, max_depth
gb_regressor.fit(X_train, y_train)

train_pred_gb = gb_regressor.predict(X_train)
test_pred_gb = gb_regressor.predict(X_test)

# Calculate RMSE
train_rmse_gb = root_mean_square_error(train_pred_gb, y_train)
test_rmse_gb = root_mean_square_error(test_pred_gb, y_test)

print("sklearn GBDT For house price:")

print("Train RMSE for GBDT is:", train_rmse_gb)
print("Test RMSE for GBDT is:", test_rmse_gb)
```

```
sklearn GBDT For house price:
Train RMSE for GBDT is: 1.2417211555288497
Test RMSE for GBDT is: 3.475175659227351
```

In [256]:
```
rf = RF()
rf.fit(X_train, y_train)

train_pred_rf = rf.predict(X_train)
test_pred_rf = rf.predict(X_test)
# print(train_pred_rf)
# print(test_pred_rf)

train_rmse_rf = root_mean_square_error(train_pred_rf, y_train)
# accu_rf = accuracy(pred_rf, y)
print("Train RMSE for rf is:", train_rmse_rf)

test_rmse_rf = root_mean_square_error(test_pred_rf, y_test)
print("Test RMSE for rf is:", test_rmse_rf)

# print("Accuracy for GBDT is:", accu_rf)
```

```
Train RMSE for rf is: 3.0152911495325685
Test RMSE for rf is: 4.114077453502243
```

In [257]:
```
# double check with sklearn
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(n_estimators=100, max_depth=5, random_s
rf.fit(X_train, y_train)
train_pred_rf = rf.predict(X_train)
test_pred_rf = rf.predict(X_test)
# print(train_pred_rf)
# print(test_pred_rf)
train_rmse_rf = root_mean_square_error(train_pred_rf, y_train)
# accu_rf = accuracy(pred_rf, y)

print("For house price:")
print("Train RMSE for RF is:", train_rmse_rf)

test_rmse_rf = root_mean_square_error(test_pred_rf, y_test)
print("Test RMSE for RF is:", test_rmse_rf)
```

```
For house price:
Train RMSE for RF is: 2.114212746223618
Test RMSE for RF is: 3.824817149581729
```

credit-g dataset

GBDT has a better performance than random forest in terms of a higher accuracy. Detailed values of accuracy can be found following each cell.

In [286]:
```
# 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_h
y = np.array(list(map(lambda x: 1 if x == 'good' else 0, y)))

from sklearn.preprocessing import LabelEncoder

# Preprocess the dataset
non_numeric = X.select_dtypes(exclude='number').columns
label_encoders = {}

for col in non_numeric:
    label_encoders[col] = LabelEncoder()
    X[col] = label_encoders[col].fit_transform(X[col])

X = X.values

# 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_siz
print(X.shape, y.shape, X_train.shape, y_train.shape, X_test.shape

# # 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_s
# print(X.shape, y.shape, X_train.shape, y_train.shape, X_test.sha
```

```
(1000, 20) (1000,) (700, 20) (700,) (300, 20) (300,)
```

In [287]:
```
X_train
```

Out[287]:
```
array([[ 1., 15.,  2., ...,  1.,  0.,  1.],
       [ 0., 45.,  3., ...,  1.,  0.,  1.],
       [ 1., 12.,  3., ...,  1.,  0.,  1.],
       ...,
       [ 2., 12.,  3., ...,  1.,  0.,  1.],
       [ 0., 24.,  1., ...,  1.,  1.,  1.],
       [ 3.,  6.,  3., ...,  1.,  0.,  1.]])
```

In [288]:
```
gb_class = GBDT()
gb_class.fit(X_train, y_train)

train_pred_gb = gb_class.predict(X_train) > 0.5
test_pred_gb = gb_class.predict(X_test) > 0.5

# Calculate RMSE
accu_train = accuracy(train_pred_gb, y_train)
accu_test = accuracy(test_pred_gb, y_test)

print("GBDT For credit-g dataset")
print("Train accu for GBDT is:", accu_train)
print("Test accu for GBDT is:", accu_test)
```

```
GBDT For credit-g dataset
Train accu for GBDT is: 0.9257142857142857
Test accu for GBDT is: 0.7733333333333333
```

In [289]:
```
rf_class = RF(n_estimators=100, max_depth=3)
rf_class.fit(X_train, y_train)

train_pred_rf = rf_class.predict(X_train) > 0.5
test_pred_rf =rf_class.predict(X_test) > 0.5

# Calculate RMSE
accu_train = accuracy(train_pred_rf, y_train)
accu_test = accuracy(test_pred_rf, y_test)

print("GBDT For credit-g dataset")
print("Train accu for rf is:", accu_train)
print("Test accu for rf is:", accu_test)
```

```
GBDT For credit-g dataset
Train accu for rf is: 0.7357142857142858
Test accu for rf is: 0.7166666666666667
```

breast cancer dataset

GBDT has a better performance than random forest in terms of a higher accuracy. Detailed values of accuracy can be found following each cell.

In [290]:
```
# 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_siz
print(X.shape, y.shape, X_train.shape, y_train.shape, X_test.shape
# X
```

```
(569, 30) (569,) (398, 30) (398,) (171, 30) (171,)
```

In [291]:
```
gb_class = GBDT()
gb_class.fit(X_train, y_train)

threshold = 0.5
gbdt_pred_train = gb_class.predict(X_train) > threshold
gbdt_acc_train = accuracy(gbdt_pred_train, y_train)

gbdt_pred_test = gb_class.predict(X_test) > threshold
gbdt_acc_test = accuracy(gbdt_pred_test, y_test)

print("GBDT classification on breast cancer dataset")
print("Train accu for GBDT is:", gbdt_acc_train)
print("Test accu for GBDT is:", gbdt_acc_test)
```

```
GBDT classification on breast cancer dataset
Train accu for GBDT is: 1.0
Test accu for GBDT is: 0.9824561403508771
```

In [308]:
```
from sklearn.ensemble import GradientBoostingClassifier as GBDT
gb_class = GBDT(n_estimators=100, learning_rate=0.9, max_depth=3,
gb_class.fit(X_train, y_train)

train_pred_gb = gb_class.predict(X_train) > 0.5
test_pred_gb = gb_class.predict(X_test) > 0.5

# Calculate RMSE
accu_train = accuracy(train_pred_gb, y_train)
accu_test = accuracy(test_pred_gb, y_test)

print("sklearn GBDT For breast cancer dataset:")
print("Train accu for GBDT is:", accu_train)
print("Test accu for GBDT is:", accu_test)
```

```
sklearn GBDT For breast cancer dataset:
Train accu for GBDT is: 1.0
Test accu for GBDT is: 0.9707602339181286
```

In [297]:
```
rf = RF()
rf.fit(X_train, y_train)


train_pred_rf = rf.predict(X_train) > 0.5
test_pred_rf = rf.predict(X_test)> 0.5
# print(train_pred_rf)
# print(test_pred_rf)

train_accu_rf = accuracy(train_pred_rf, y_train)
test_accu_rf = accuracy(test_pred_rf, y_test)

# accu_rf = accuracy(pred_rf, y)
print("RF classification on breast cancer dataset:")
print("Train accuracy for rf is:", train_accu_rf)

print("Test accuracy for rf is:", test_accu_rf)
```

```
RF classification on breast cancer dataset:
Train accuracy for rf is: 1.0
Test accuracy for rf is: 0.9590643274853801
```

In [298]:
```
from sklearn.ensemble import RandomForestClassifier as RF
# from sklearn.preprocessing import LabelEncoder

# label_encoder = LabelEncoder()
# y_train_encoded = label_encoder.fit_transform(y_train)
# y_test_encoded = label_encoder.transform(y_test)

rf = RF(n_estimators=100, max_depth=5, random_state=42)
rf.fit(X_train, y_train)

train_pred_rf = rf.predict(X_train) > 0.5
test_pred_rf = rf.predict(X_test) > 0.5
# print(train_pred_rf)
# print(test_pred_rf)
train_accu_rf = round(accuracy(train_pred_rf, y_train),2)
test_accu_rf = round(accuracy(test_pred_rf, y_test),2)

print("sklearn RF for breast cancer dataset:")

print("Train accu for rf is:", train_accu_rf)
print("Test accu for rf is:", test_accu_rf)
```

```
sklearn RF for breast cancer dataset:
Train accu for rf is: 0.99
Test accu for rf is: 0.95
```

GBDT captures complex relationships within data compared to Random Forest. Since GBDT builds trees and continiously correct errors made by previous ones, it can capture more complexity. This characteristic can potentially be more accurate when having a complex dataset.
But GBDT minimizes loss function using gradient descent and predict iteratively, which can lead to better performance, especially with complex data.

Random Forest builds multiple trees independently, leading to poorer response to complex data relationship