1

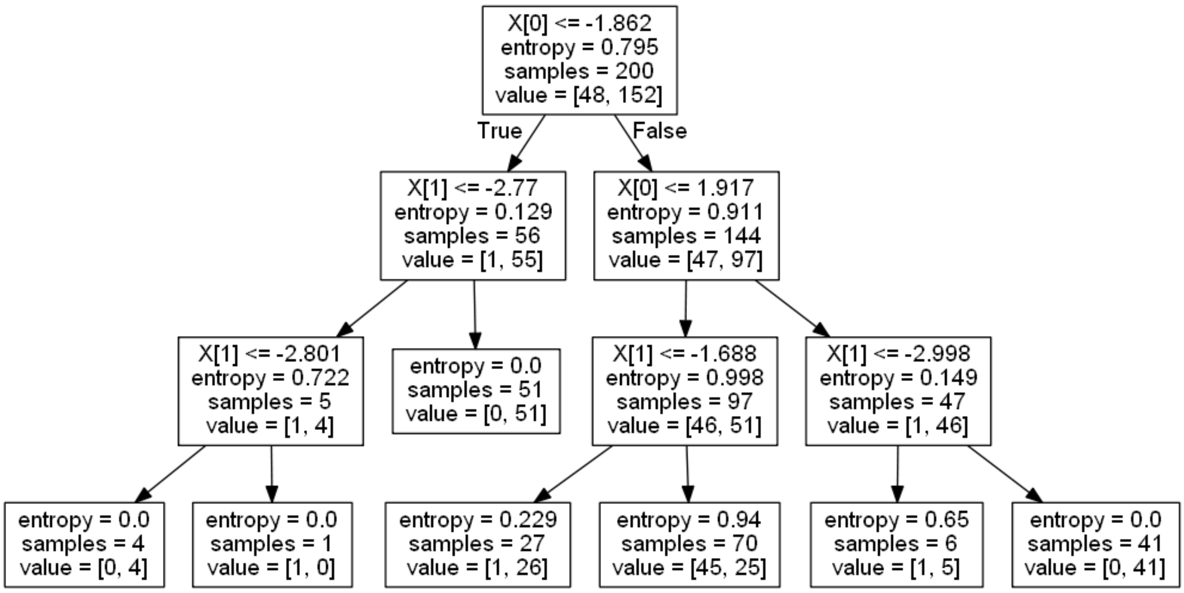
```
def compute_entropy(label_array):
    label_array = np.array(label_array).flatten()
    n_classes = np.unique(label_array)
    entropy = 0
    length = len(label_array)
    for label in n_classes:
        countDict = Counter(label_array)
        p = countDict[label]/length
        entropy += -p*np.log(p)
    return entropy

def compute_gini(label_array):
    label_array = np.array(label_array).flatten()
    n_classes = np.unique(label_array)
    gini = 0
    length = len(label_array)
    for label in n_classes:
        countDict = Counter(label_array)
        p = countDict[label]/length
        gini += p*(1-p)
    return gini
```

2

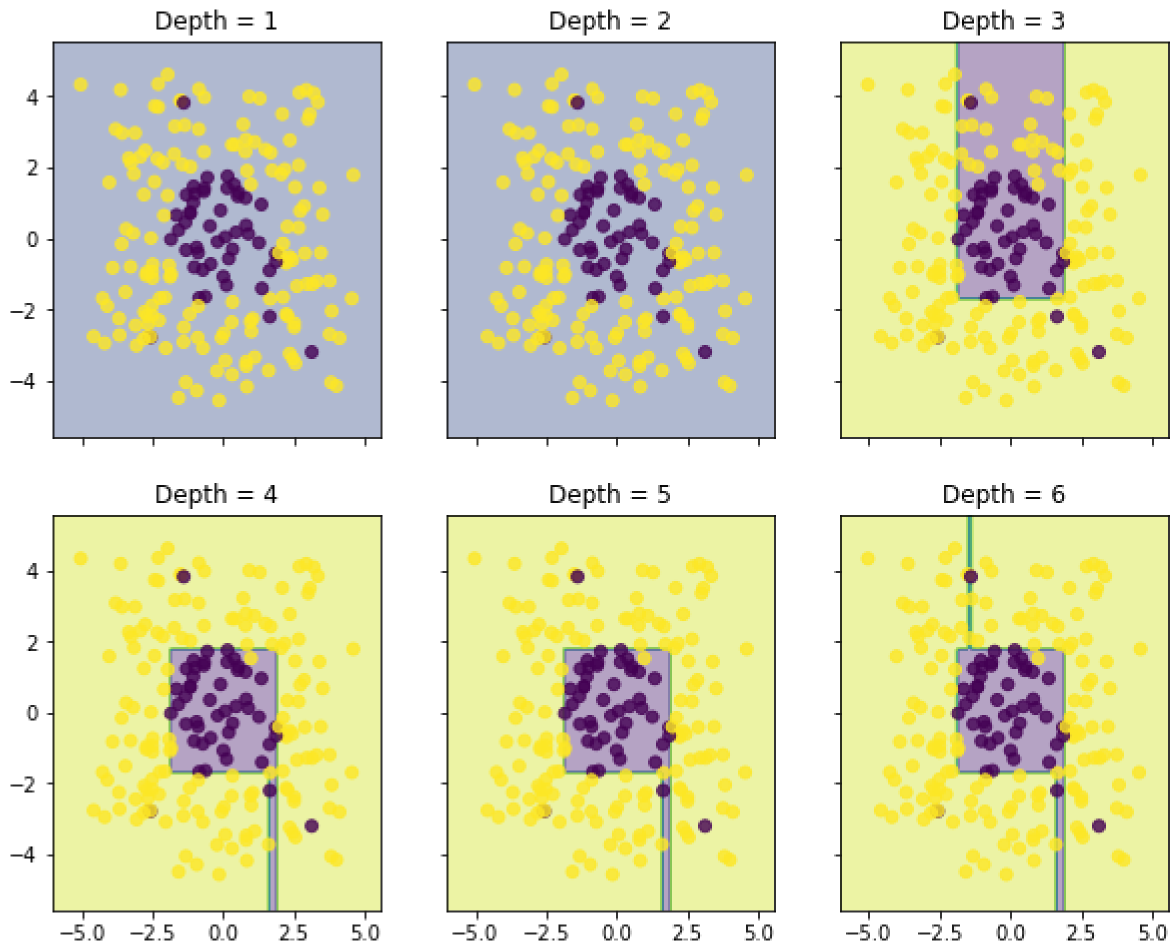
```
class Decision_Tree(BaseEstimator):

    def __init__(self, split_loss_function, leaf_value_estimator,
                 depth=0, min_sample=5, max_depth=10):
        self.split_loss_function = split_loss_function
        self.leaf_value_estimator = leaf_value_estimator
        self.depth = depth
        self.min_sample = min_sample
        self.max_depth = max_depth
        self.is_leaf = True
        # Node Info
        self.value = None
        self.split_id = None
        self.split_value = None
        self.split_index = None
        self.left = None
        self.right = None

    def fit(self, X, y):
        n, m = X.shape
        if self.depth == self.max_depth or n <= self.min_sample:
            self.value = self.leaf_value_estimator(y)
            return self

        best_loss = self.split_loss_function(y)
```

```
        best_split_index = None
        best_split_id = None
        combine = np.concatenate([X, y], axis=1)
        for i in range(m):
            combine = np.array(sorted(combine, key=lambda x: x[i]))
            y = np.array(combine[:, -1]).reshape(-1, 1)
            split_index, loss = self.find_best_split(y)
            if loss < best_loss:
                self.is_leaf = False
                best_loss = loss
                best_split_index = split_index
                best_split_id = i

        if self.is_leaf:
            self.value = self.leaf_value_estimator(y)
        else:
            combine = np.array(sorted(combine, key=lambda x: x[best_split_id]))
            X = np.array(combine[:, :-1])
            y = np.array(combine[:, -1]).reshape(-1, 1)
            self.split_id = best_split_id
            self.split_value = X[best_split_index - 1, best_split_id]
            self.split_index = best_split_index - 1
            self.left = Decision_Tree(self.split_loss_function,
self.leaf_value_estimator, self.depth + 1, self.min_sample, self.max_depth)
            self.right = Decision_Tree(self.split_loss_function,
self.leaf_value_estimator, self.depth + 1, self.min_sample, self.max_depth)
            self.left.fit(X[:best_split_index], y[:best_split_index])
            self.right.fit(X[best_split_index:], y[best_split_index:])
        return self

    def find_best_split(self, y):
        n = y.shape[0]
        best_loss = np.inf
        split_index = 0
        for i in range(n - 1):
            loss_left = (i + 1) * self.split_loss_function(y[:i + 1]) / n
            loss_right = (n - i - 1) * self.split_loss_function(y[i + 1:]) / n
            loss_total = loss_left + loss_right
            if loss_total < best_loss:
                best_loss = loss_total
                split_index = i + 1
        return split_index, best_loss

    def find_best_feature_split(self, X, y):
        # The code is in the fit method, this function is redundant, it call
        # find_best_split but cannot return the best loss, and best split value
        pass

    def predict_instance(self, instance):
        if self.is_leaf:
            return self.value
        if instance[self.split_id] <= self.split_value:
            return self.left.predict_instance(instance)
        else:
            return self.right.predict_instance(instance)
```

3



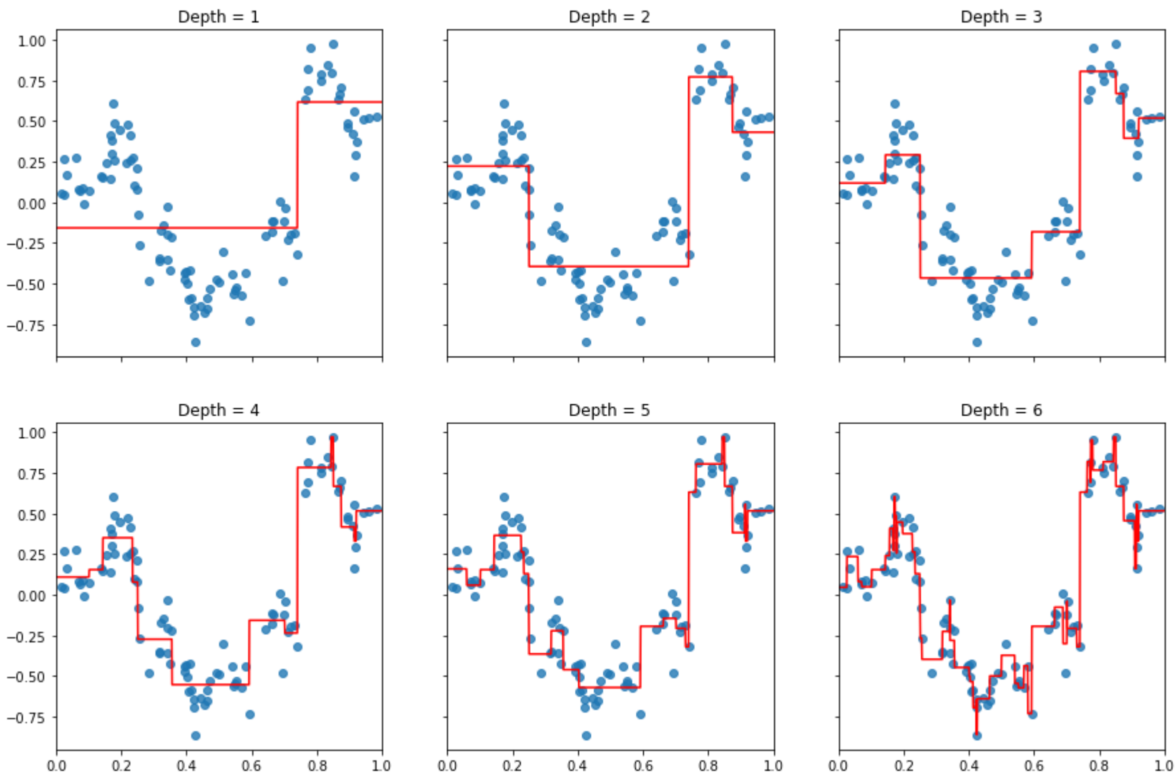
X[0] <= -1.862
entropy = 0.795
samples = 200
value = [48, 152]
True
False
X[1] <= -2.77
entropy = 0.129
samples = 56
value = [1, 55]
X[0] <= 1.917
entropy = 0.911
samples = 144
value = [47, 97]
X[1] <= -2.801
entropy = 0.722
samples = 5
value = [1, 4]
entropy = 0.0
samples = 51
value = [0, 51]
X[1] <= -1.688
entropy = 0.998
samples = 97
value = [46, 51]
X[1] <= -2.998
entropy = 0.149
samples = 47
value = [1, 46]
entropy = 0.0
samples = 4
value = [0, 4]
entropy = 0.0
samples = 1
value = [1, 0]
entropy = 0.229
samples = 27
value = [1, 26]
entropy = 0.94
samples = 70
value = [45, 25]
entropy = 0.65
samples = 6
value = [1, 5]
entropy = 0.0
samples = 41
value = [0, 41]



Depth = 1
Depth = 2
Depth = 3
Depth = 4
Depth = 5
Depth = 6
4
2
0
−2
−4
−5.0
−2.5
0.0
2.5
5.0

4

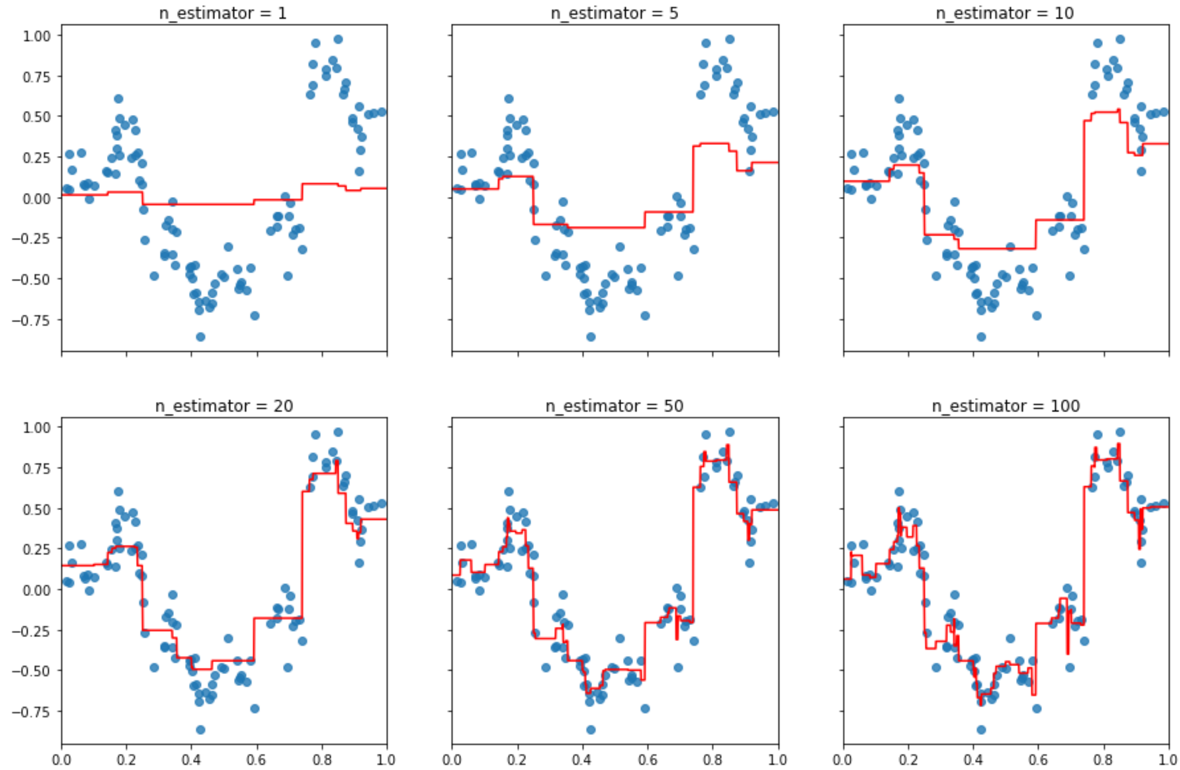
```python
def mean_absolute_deviation_around_median(y):
    median = np.median(y)
    mae = np.mean(np.abs(y - median))
    return mae
```



5

```
class gradient_boosting():
    def __init__(self, n_estimator, pseudo_residual_func, learning_rate=0.01,
                 min_sample=5, max_depth=5):
        self.n_estimator = n_estimator
        self.pseudo_residual_func = pseudo_residual_func
        self.learning_rate = learning_rate
        self.min_sample = min_sample
        self.max_depth = max_depth

        self.estimators = []  # will collect the n_estimator models

    def fit(self, train_data, train_target):
        X, y = train_data, train_target
        n, m = train_data.shape
        for _ in range(self.n_estimator):
            y_acc = np.zeros(n)
            for i in range(len(self.estimators)):
                for j in range(n):
                    y_acc[j] += self.learning_rate * \
                        self.estimators[i].predict_instance(X[j])
            rt = Regression_Tree(
                min_sample=self.min_sample, max_depth=self.max_depth,
loss_function='mse', estimator='mean')
            rt.fit(X, self.pseudo_residual_func(y, y_acc).reshape(-1, 1))
            self.estimators.append(rt)

    def predict(self, test_data):
        X = test_data
        test_predict = np.zeros(len(X))
        for i in range(self.n_estimator) :
            for j in range(len(X)) :
                test_predict[j] += self.learning_rate *
self.estimators[i].predict_instance(X[j])
        return test_predict
```

6



7

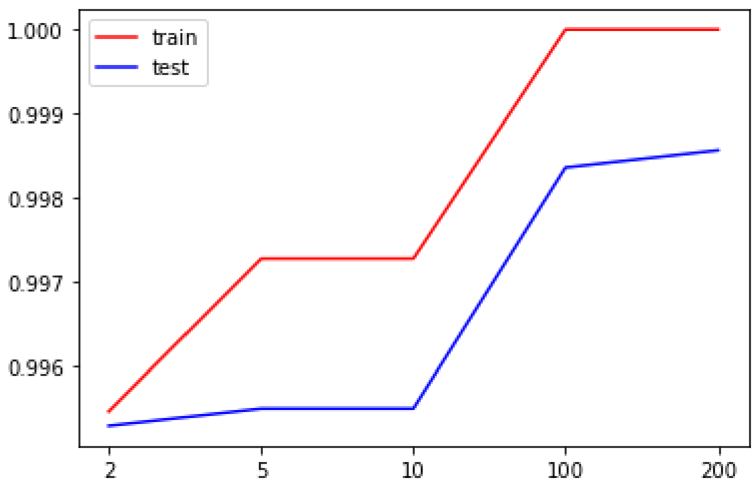
$$r_i = \frac{y_i}{1 + e^{y_i f(x_i)}}$$

Dimension is m-1

8

$$h_m = argmin \sum_{i=1}^{n} [\frac{y_i}{1 + e^{y_i f_{m-1}(x_i)}} - h(x_i)]$$

9

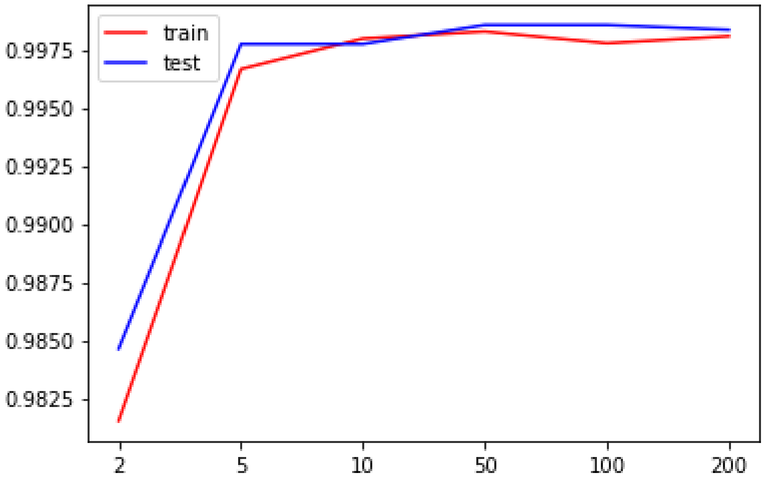


10

Random Forest generate multiple decision tree

1. For each tree, select(bagging) a subset of feature(~$\sqrt{m}$) to build DT
2. make decision by gather(ensamble) all trees' result(ex. mode)

11



12

When the amount of features selected by RF are very high, it may overfitting. Gradient Boosted Trees achieve higher training accuracy, but it is more sentive to outliers and easy to overfit(compare to RF), it is not our expect. It cannot run parallel(compare to DT and RF), so it would be much slower. Random Forest performs are more generalize and better in test accuracy.